COMPUTER SCIENCE EDUCATION

SECOND EDITION 2 SECOND EDITION

ONLINE RESOURCES

PERSPECTIVES ON TEACHING AND LEARNING IN SCHOOL

EDITED BY
SUE SENTANCE,
ERIK BARENDSEN,
NICOL R. HOWARD, AND
CARSTEN SCHULTE

BLOOMSBURY

Computer Science Education

ALSO AVAILABLE FROM BLOOMSBURY

Education and Technology, 3rd edition, Neil Selwyn
Reflective Teaching in Secondary Schools, Andrew Pollard and Caroline Daly with Katharine Burn, Aileen Kennedy, Margaret Mulholland, Jo Fraser-Pearce, Mary Richardson, Dominic Wyse and John Yandell
Reflective Teaching in Primary Schools, Andrew Pollard and Dominic Wyse with Ayshea Craig, Caroline Daly, Sarah Seleznyov, Sinead Harmey, Louise Hayward, Steve Higgins and Amanda McCrory
The Art Teacher's Guide to Exploring Art and Design in the Community, Ilona Szekely
Special Educational Needs and Disabilities in Schools, 2nd edition, Janice Wearmouth
Teaching Personal, Social, Health and Economic and Relationships, (Sex) and Health Education in Primary Schools, edited by Victoria Pugh and Daniel Hughes
Education for Social Change, Douglas Bourn
Why Do Teachers Need to Know About Diverse Learning Needs?, edited by Sue Soan
Why Do Teachers Need to Know About Psychology?, edited by Jeremy Monsen, Lisa Marks Woolfson and James Boyle
Why Do Teachers Need to Know About Child Development?, edited by Daryl Maisey and Verity Campbell-Barr

Computer Science Education

Perspectives on Teaching and Learning in School

Edited by
Sue Sentance, Erik Barendsen, Nicol R. Howard and Carsten Schulte

BLOOMSBURY ACADEMIC
LONDON • NEW YORK • OXFORD • NEW DELHI • SYDNEY

BLOOMSBURY ACADEMIC
Bloomsbury Publishing Plc
50 Bedford Square, London, WC1B 3DP, UK
1385 Broadway, New York, NY 10018, USA
29 Earlsfort Terrace, Dublin 2, Ireland

BLOOMSBURY, BLOOMSBURY ACADEMIC and the Diana logo are trademarks of Bloomsbury Publishing Plc

First published in Great Britain 2018

This edition published 2023

Cover design by Charlotte James
Cover image: Donald Iain Smith / Getty Images

A catalogue record for this book is available from the British Library.

A catalog record for this book is available from the Library of Congress.

ISBN: HB: 978-1-3502-9691-6
PB: 978-1-3502-9690-9
ePDF: 978-1-3502-9692-3
eBook: 978-1-3502-9693-0

Typeset by Newgen KnowledgeWorks Pvt. Ltd., Chennai, India

Online resources to accompany this book are available at Bloomsbury.pub/computer-science-education-2e. If you experience any problems, please contact Bloomsbury at: onlineresources@bloomsbury.com

Contents

How to Use This Book

The book provides a resource for all those studying computer science education, whether as part of a teacher training programme, or as part of postgraduate study, as well as for computer science teachers wishing to understand their subject better. The book explores why and how computer science can be taught effectively in schools. It is not country- or curriculum-specific so will support you wherever you are working in the world.

As well as summarising key theories and research development in the field in an accessible way, the book provides opportunities throughout the chapters to engage with real-life examples from practice, and key concepts and questions for further reflection.

Within Each Chapter

Chapter Synopsis Each chapter begins with a chapter synopsis which summarises the central focus of the chapter.

Examples Within each chapter are examples that illustrate the material being covered in the chapter. In some chapters, these are practical examples of activities that you could carry out in your practice. In other chapters, these are examples from research that illustrate theoretical points to illuminate them.

Key Concepts Key concepts are drawn out of the main text and explained in more detail. Highlighting key terms enables you to ensure you understand the concept and can define it.

At The End Of Each Chapter

Key points The key points from each chapter are summarised in a short set of bullet points to highlight key takeaways.

Further Reflection Each chapter ends with an opportunity for further reflection through questions. This enables you as a reader to reflect upon your own practice and experience and to begin to think about how this might impact on your future practice.

Figures

Tables

Contributors

Editors

Sue Sentance is Director of the Raspberry Pi Computing Education Research Centre at the University of Cambridge, UK, and the Chief Learning Officer at the Raspberry Pi Foundation. She has been involved in the curriculum changes surrounding computing in England for some years, and her research areas include the use of PRIMM to structure lessons, productive classroom talk and teacher professional development.

Erik Barendsen is Professor of Computing Education at Open University and Professor of Science Education at Radboud University, the Netherlands. His scientific interests include design-based and context-based teaching and learning in computer science and STEM subjects, computational thinking and its integration into the school curriculum, digital literacy, teachers' practical knowledge, in particular pedagogical content knowledge and teachers' professional development.

Nicol R. Howard is Associate Professor and Associate Dean of Academic Affairs in the School of Education at the University of Redlands, USA. She is also a co-director of the Race in Education Analytics Learning Lab (REAL Lab) at the Center for Educational Justice. Her research focuses on equity in STEM and computer science education as well as community and family involvement.

Carsten Schulte is Professor at Paderborn University, Germany, and head of the computer science education research group. His main research interests include the conceptualization of computing education for the digital world, exploring computer biographies, educational modelling of program comprehension and developing ideas and concepts for data science and AI education.

Contributors

Timo Arnken is a teacher at Altes Gymnasium Oldenburg, Germany, where he leads informatics across the school.

Tim Bell is Full Professor in the Department of Computer Science and Software Engineering at the University of Canterbury in Christchurch, New Zealand.

Lea Budde is Research Associate in Didactics of Informatics at the University of Paderborn, Germany.

Shannon Campe is Research Associate and Project Coordinator at Education, Training, Research, USA.

Michael Caspersen is Managing Director of It-vest and Honorary Professor at the Department of Computer Science at Aarhus University, Denmark.

Layote Chandler is a doctoral student of educational technology at the University of Florida, USA.

Alexis Cobo is a doctoral student of educational technology at the University of Florida, USA.

Paul Curzon is Professor of Computer Science in the School of Electronic Engineering and Computer Science at Queen Mary University of London, UK.

Quintin Cutts is Professor of Computer Science Education within the School of Computing Science at the University of Glasgow, UK.

Jill Denner is Senior Research Scientist at Education, Training, Research, USA.

Ira Diethelm is Professor of CS Education at University of Oldenburg, Germany.

Peter Donaldson is University Lecturer in Computing Science and Education at the University of Glasgow, UK.

James Donohue was a member of the Learner Development Team, at Manchester Metropolitan University, UK, until May 2021.

Caitlin Duncan was a member of the Computer Science Education Research Group at the University of Canterbury, New Zealand, and is a postdoctoral fellow at the University of Osaka, Japan.

Juliana Goschler is Professor of German as a Second/Foreign Language, focusing on the use of language and terminology in different subjects at school to support learning.

Shuchi Grover is Senior Research Scientist at Looking Glass Ventures and a visiting scholar at Stanford University, USA.

Sven Hüsing is Research Associate in Computer Science Education within the Project Data Science and Big Data at School (ProDaBi) at Paderborn University, Germany.

Maya Israel is Associate Professor of Educational Technology and Computer Science Education in the School of Teaching and Learning at the University of Florida, USA.

Ilkka Jormanainen is Senior Researcher in the Technologies for Learning and Development Research Group at the School of Computing, University of Eastern Finland, Finland.

Juho Kahila is a doctoral student in the School of Applied Educational Science and Teacher Education, University of Eastern Finland, Finland, focusing on children's learning related to digital games and metagame.

Maria Kallia is Research Associate within the School of Computing Science (Centre for Computing Science Education) at the University of Glasgow, UK.

Tia C. Madkins is Assistant Professor in the College of Education at the University of Texas at Austin, USA.

William Marsh is Senior Lecturer in the School of Electronic Engineering and Computer Science at Queen Mary University of London, UK.

Peter W. McOwan† was Professor of Computer Science in the School of Electronic Engineering and Computer Science at Queen Mary University of London, UK.

Roy Pea is David Jacks Professor of Education and Learning Sciences at Stanford University, School of Education, USA, and director of the H-STAR Institute.

Mara Saeli is Assistant Professor of Computer Science Education at Radboud University, Nijmegen, the Netherlands.

Juha Sorva is Senior University Lecturer in the Department of Computer Science at Aalto University, Finland.

Matti Tedre is Professor in the School of Computing at the University of Eastern Finland, Finland.

Tapani Toivonen is University Lecturer at the University of Eastern Finland, Finland.

Teemu Valtonen is Professor of Education at the University of Eastern Finland, Finland.

Henriikka Vartiainen is University Lecturer and Researcher at the University of Eastern Finland, Finland.

Jane Waite is Senior Research Scientist at the Raspberry Pi Foundation and a doctoral student at Queen Mary University of London, UK.

Lauren Weisberg is a doctoral student in Curriculum and Instruction at the University of Florida, USA.

Felix Winkelnkemper is Research Associate in the Computer Science Education Research group at the University of Padeborn, Germany.

Seymour Wright is a saxophonist and independent researcher – his work is focused on the nexus of group learning, improvisation and creative practice.

Preface

This book is the result of many, many conversations with teachers, researchers and teacher educators in many different countries – all engaged in the broad field of computer science education. It is an attempt to bridge the gap between practical and country-specific 'how-to' books on teaching computer science/informatics/computing (different names for our subject exist!) and the growing body of computer science education research pertaining to schools. The goal of the book was to bring together key experts in the field to explain their areas of research and its relevance in the classroom, in an accessible way, whilst retaining enough depth to be useful and provide a basis for practitioners to follow and engage in reflection and eventually even research on CS education. We hope that in this volume we have managed to do this and that our book will be useful to those training to teach computer science, as well as those already teaching computer science who wish to understand the issues in more depth.

This book specifically refers to school education (known as K–12 in the United States and elsewhere), rather than computer science in higher education, and will also be useful in identifying key areas of computer science education research relevant to bringing the subject into the school curriculum, as many countries are now doing.

The original text was published in 2018, and we are delighted to be able to publish an updated edition. As well as no less than thirty-five contributors, plus the editors, highlights of this second edition include the following:

- New sections on machine learning and (data-driven) epistemic programming
- A new focus on equity and inclusion in computer science education
- A revised chapter on relating ethical and societal aspects to knowledge-rich aspects of computer science education
- An extended set of chapters on the learning of programming, including design, pedagogy and misconceptions

In addition, we have managed to retain nearly all the content from the first edition! This book represents perspectives from all over the world and is intended to be accessible to you wherever you live and work. It does not relate to any particular country or curriculum, although some countries have been used as examples at times to illustrate key points. Although approaches to teaching computer science in school vary considerably in different countries, the key issues remain the same. We hope that drawing on a wide variety of perspectives has made this book even more valuable.

We hope you enjoy reading this book and find much to enhance your teaching!

Foreword to the Second Edition

Across the world a major shift is taking place in computing education at school: children will in future learn computer science as a foundational discipline, from primary school onwards, just as they do mathematics and natural science. This change reflects a seismic shift, away from regarding computing merely as a technology that we must all grapple with and towards thinking of it as a subject discipline in its own right.

But once we say 'we should treat computer science as foundational, for every child, not only the software developers of the future', we must confront the deep questions: what should we teach, and how should we teach it? Other subject disciplines have had centuries to develop answers to these questions (and are still debating them), but computing has not. Yet the need is pressing, because educators across the world are hungry for an inspiring vision to frame their teaching, for pedagogies that provably work and for assessments that validly measure learning and progress.

So this book is extremely timely. It tackles both of the big questions – the 'what' and the 'how' – and does so from multiple perspectives. Moreover, because computer science education at the school level is still so new, it is hugely important that classroom practice is informed and guided by evidence rather than gut feelings. This book clearly meets that bar: it is written by many of the leading researchers and experts in computing education across the world.

But it is equally important that research should in turn be focused on the realities of the classroom. The book is written in a language that a classroom teacher, or a student training to be a teacher, can make sense of. Indeed, these teachers are the audience this book is intended to serve.

This second edition has a whole new section on equity and inclusion, which I particularly welcome in a discipline that, however inadvertently, skews heavily white and male. This is a lost opportunity that isn't going to fix itself; we need to pay sustained attention to it.

Nobody has a monopoly on truth. Really good education is hard, and we are all feeling our way as we seek to inspire our young people with the joy and beauty of computer science. But the authors of this book have spent their professional lives studying computing in the classroom, and I am absolutely delighted to see such a substantial contribution in such an underserved space. Enjoy!

Professor Simon Peyton Jones OBE
Epic Games & University of Cambridge, UK

Part 1

Positioning Computer Science in Schools

1

Introduction to Part 1

Carsten Schulte

Computer science teachers teach computer science. So much is obvious, but what does that actually mean? What does computer science encompass? Why should it be taught in schools at all, and, to be more concrete, what is it that should be taught, why should it be taught and for whom? This first section of the book is targeted at such fundamental questions. Its chapters provide a range of perspectives on teaching and learning computer science at a school level. Their common goal is to provide a basis which informs the discourse about the role of the subject in the context of general education. In addition, they demonstrate the multifaceted nature of computer science and its interaction with other disciplines (including arts and humanities) and also its ethical and social implications.

Chapter 2 by Matti Tedre highlights the subject of computer science itself from three distinct ways to understand the subject matter. Does computer science in form and style resemble mathematics? Is it a discipline of engineering? In what way is it like natural science, or might it be something entirely different?

In Chapter 3, Erik Barendsen and Mara Saeli look into existing computer science curricula and highlight their multifaceted status. While from a government and societal point of view, they represent political appreciations of the importance of computer technology as a whole and of certain aspects in particular, for practitioners on a school level, they provide guidelines and indications for the planning of lessons without being definitive as what is written down in curricula often is not the same as what is actually implemented in the classroom.

Chapter 4 by Carsten Schulte, Felix Winkelnkemper and Lea Budde aims at an integration of different perspectives on the subject matter. While traditionally, computer science education puts a lot of emphasis on the architectural aspects and their theoretical underpinning, the chapter argues that in order to actual understand what products of computing are, what their role in the world is and why they got the way they are, one has to integrate architectural knowledge with relevant discourses, artefact genesis and the roles users and stakeholders have in the interaction with them.

In Chapter 5, Shuchi Grover and Roy Pea describe computational thinking as an approach which makes the way computer scientists think and tackle problems relevant to general education. The chapter explains what this encompasses and how the new competency changes both computer science courses and how it can be integrated with other subject courses.

Chapter 6 by Ilkka Jormanainen, Matti Tedre, Henriikka Vartiainen, Teemu Valtonen, Tapani Toivonen and Juho Kahila explains how to learn machine learning (ML). It characterizes ML as a new way to tackle problems through computing which, while before it was based on explicitly programmed rules, now becomes data driven. What such a change means in terms of its application in K–12 schools is explored in detail using real-life examples.

2

The Nature of Computing as a Discipline

Matti Tedre

Chapter outline

Chapter synopsis

What is computer science? What should we teach about computing and how? Which skills and knowledge are central to computing? Over the disciplinary history of computing, there has never been a consensus on the field's fundamental nature, aims, methods, essential skills and knowledge, its relations to other disciplines or even the name of the field. In the course of their development, computing education initiatives such as curriculum development (see Chapter 3 in this book), course design and study program design often get to a point where the stakeholders start to question their consensus on how exactly do people define their field of study. Debates on those questions have characterized computing education discussions ever since the birth of the field, and many disagreements still remain. This chapter uses a historical perspective to introduce the reader to what is at stake in debates of computing's disciplinary nature, what the central positions are and how those positions differ from each other.

2.1 Introduction: Computing as a discipline

Teachers of any subject have an explicit or implicit idea of the essence of their field. Each teacher has a view and opinion of, for instance, their discipline's subject matter, aims, fundamental questions, methods and the most important achievements. Those ideas guide them when they teach their subject, develop courses, design curricula or engage in education in other ways. This is just the same in computing. We have our perceptions of computing as a discipline, as a profession and as a body of knowledge. Through our teaching, we impart some of our own ideas to students, affecting how students learn to perceive their field.

Over the years, pioneers of computing have characterized the field of computing in a great number of ways. Some argue that computing is a branch of mathematical logic, and others argue that computing is a design and engineering field. Some emphasize computing's scientific nature, while others its constructive character. All those arguments have been used to model computing education in different ways. Of the hundreds of characterizations of the field, some are more popular or influential than the others, but there is no 'correct' interpretation of computing's disciplinary nature: different views are justified from different perspectives. Popularity of different views has also changed over the years with computing's evolving status in the university as well as its developing technological state-of-the-art and theoretical body of knowledge.

One characterization of computing's disciplinary nature is the report 'Computing as a Discipline' (Denning et al., 1989), which was commissioned to support new joint curriculum recommendations by two major organizations in computing: the Association for Computing Machinery (ACM) and the IEEE Computer Society. That report describes computing as a combination of three intertwined traditions: theory, modelling and design. These traditions derive from three disciplinary lenses: the analytical, scientific and engineering lenses. The first one is theoretically oriented and emphasizes formal methods of mathematics and logic. The second one is empirically oriented and features data, simulation and abstraction. The third one is technologically oriented and emphasizes design and engineering methods. The three traditions have their own aims and goals, and their differences have practical ramifications. They differ in terms of their methods, assumptions, views of knowledge, perceptions of the structure of reality, concepts of human nature and world view in general.

The 'Computing as a Discipline' report started to greatly affect how computing educators viewed their field. Most computing teachers indeed know some theory, do some design and engage in modelling or abstraction activities. But although those activities support each other, the three traditions of computing are also profoundly different from each other and in many ways incompatible.

In order to be able to give students a balanced and rich view of computing as a discipline, it is important to understand these different traditions, their research agendas and their roots. Viewing computing from a variety of perspectives offers educators meaningful entry points to computing's topics, and it offers students insights into the immense theoretical, practical, scientific and philosophical richness of computing.

This chapter introduces three dominant traditions of computing and characterizes their aims, questions, views of subject matter, methods and practices. These traditions are surely not the only ones at play, as one could easily argue that computing borrows many elements from social sciences and human sciences or that it might best be described as an interdisciplinary field. However, as most of computing's disciplinary debates boil down to its theoretical, scientific and engineering features, this chapter focuses on these three.

2.2 Computing is a field of engineering and design

Those who view computing as primarily a technological field, or a field of engineering and design, have a strong case for their argument. Computing's development as an independent discipline started only after the birth of the modern computer. Computing's history reveals a rich collection of technical and technological breakthroughs, computing machinery and a union of theory and craftsmanship. Many pioneers of modern computing, such as Bush, Eckert and Atanasoff, were electrical engineers. The histories of automatic office machinery, scientific instruments, calculating instruments and military equipment are woven in the fabric of modern computing. Progress of computing has been driven by increasingly smarter designs as well as technological tendencies, such as the exponential growth rate of transistors in chips (Moore's law), the even faster growth of memory technology density (Kryder's law) and the exponential growth of communication speed.

For decades, engineering and design were, however, downplayed in discussions about academic computing. Even though software technology and system design were important drivers of progress – and often developed in universities – they did not resonate well with computing's campaign to achieve an independent disciplinary identity in the academia. The issue was not that computer systems and software design would not be important, but it was about the rigor and academic image of the aspiring discipline of computing. There was a widespread opinion among the more established disciplines that design and engineering were not well-suited for traditional research universities. So, in their quest for making computing an independent discipline, some computing pioneers proudly promoted a view of computing as an abstract field that downplayed everything that had to do with designing, building and developing computer systems, societally valuable applications and computing machinery. But however intellectually justified, their views of computing as just a branch of mathematics and logic were far from what was really happening in computing practice and much of computing research.

The design and engineering aspects of computing were revived in the end of the 1960s, when a thought-provoking term *software engineering* was promoted as a solution to the sad state of software in the time of the so-called software crisis. The software engineering movement quickly gained momentum, as many practitioners and software-oriented academic people felt that their work is much better characterized by engineering and design than mathematics and logic. Over some three decades of development, software engineering matured into a progressive field and became a part of computing's core knowledge.

Proponents: We are designers and builders

Computing's engineering nature has been described and justified in many ways. Some proponents of computing's engineering character look at computing practice and argue that instead of mathematics or science, the majority of computer scientists – both practitioners and researchers – actually do engineering, design and development. Others base their argument on methods and outcomes and argue that engineering approaches are required for reaching the solid reliability and safety record of rigorous engineering design in other fields. Yet others wish designers to achieve the same sort of intellectual recognition and triumphant image that engineering had in the first half of the twentieth century. Those who argue that engineering describes best what people in computing actually do often justify their argument by looking at the aims and central questions, necessary skill set and methods and practices of computing.

Aims and questions

Frederick P. Brooks Jr, a computing pioneer and author of several classic works on software engineering, made a distinction between a scientist and an engineer: the scientist builds in order to study, and the engineer studies in order to build (Brooks, 1996). Similarly, while natural sciences focus on 'what' type of questions, some argue that computing's focus on 'how' type of questions reveals its engineering character (Hartmanis, 1994). Hartmanis (1994) wrote that whereas natural scientists ask 'what exists?', computer scientists ask 'what can exist?' The engineering view sees that many problems in contemporary computing are not *whether* a program, algorithm, technique or system can exist, but *how* to make one in practice.

Many people also argue that the aim of computing has, from the very beginning, been to design and construct useful things. Hence, similar to any other engineering field, research in computing is about development of methods and tools that advance the state-of-the-art or enable new things to be done. Unlike natural scientists who deal with naturally occurring phenomena, engineers deal with artefacts that are created by people; for many people, that makes computing an engineering field. And while knowledge in natural sciences progresses through experiments, in computing, 'demonstrations' is the keyword: in many computing fields, the scientists' slogan 'publish or perish' has changed into the engineers' slogan 'demo or die' (Hartmanis, 1994).

Subject matters

Those who argue that computing is essentially an engineering field have a difficult question to answer: if it is engineering, what is it the engineering *of*? If chemical engineering is the application of chemistry and mechanical engineering is the application of material science and physics, what is computing, as an engineering field, the application *of*? Attempts to address the 'engineering of' issue include, for example, engineering of mathematics (or mathematical processes), information engineering, cognitive technology, conceptual engineering, language of technology and mechanization of abstraction (Tedre, 2014). None of these has gathered any widespread support.

From the subject matter viewpoint, the engineering view of computing boils down to two key issues: artifice and causality. Firstly, many engineering-oriented researchers and developers in

computing pose problems and follow the design process to come up with a solution in the form of an *artefact* (Denning et al., 1989; Simon, 1969). Different from natural sciences, which deal with naturally occurring things, computing deals with artefacts, which were designed for certain purposes in specific contexts of use. And different from mathematics, which deals with abstract objects, executable programs of computing are causal, physical things: they are swarms of electrons in the circuits of a computer. When run, they can cause change in the world. Computer programs can make monitors blink and printers rattle; they can drive cars and guide missiles to their targets. Suggestions by the engineering camp for computing's subject matter include, for example, software, computer systems and computers.

Methods and practices

Some argue that the necessary skill set for computing reveals the field's engineering nature. Design, in particular, is essential to computing, and in line with that, programming has been described as an art, craft, trade or skill. Many pioneers of computing point out that unlike research in many traditional theoretical and empirical sciences, development of working computer systems has to cater much more to material resources, social and human constraints, budgets as well as laws of nature. Many people in computing design complex systems with limitations to resource consumption. Design involves studying users, groups of users and communities, which requires methods from social sciences and humanities. And like engineers in other fields, programmers follow a systematic sequence of design decisions to exclude alternative options until a solution is ready (Wegner, 1976).

Examples: Design view in schools

Countless initiatives have introduced computing in schools through designing and building artefacts – such as building and programming educational robots, designing and creating games without writing code and creating apps for smartphones. At the university level, summative assessment is often based on students designing and implementing a solution to meet a concern, answer a threat or rise to an opportunity. All of these are design and engineering activities.

Opponents: Building things is not an academic aim

In addition to passionate supporters, the engineering view of computing has also strong opponents. In their attempt to improve computing's academic status, many pioneers of computing downplayed computing's constructive character and highlighted its theoretical aspects (Tedre, 2014). Early software engineering was accused of sloppy methodology and lack of rigorous theories. Some pointed out the short lifespan of technical inventions: as a fundamental discipline, computing should focus on enduring fundamental principles instead of technological solutions that obsolete quickly (Wegner, 1970). Many theoretically oriented pioneers of computing wanted to see

computing focus on what is common to the use of any computer in any application instead of technical details or societal aspects of computing.

Accusations of lack of rigor in software construction and research undermined computing's engineering image. Those claims were first anecdotal and based on people's subjective perceptions of how software construction was done. Many people pointed out the undeveloped theoretical foundations of software engineering, its scarcity of theoretically and technically well-developed methods and that it seemed to be guided by rules of thumb, toying and tinkering. Later, in the 1990s, systematic reviews of publications in software engineering indeed showed a lack of attention to methodology and theory in software engineering publications.

However, some zealots aside, the usual argument against the engineering view was never about engineering and design being unimportant. Producing useful and reliable systems has always been a societally important aim that poses endless intellectual challenges. Instead, the critical voices about seeing computing as a branch of engineering or technology are about the lasting value of engineering and design, its contribution to our common knowledge about the world and its centrality in the *academic* discipline of computing and in computing research.

Key concept: Computing as engineering

Many computing activities are centred around requirement analysis, design, implementation, evaluation and production of useful artefacts. The questions that computing answers, and the problems it solves, are often of the engineering-type 'how' questions. Many problems in computing require design and engineering methods and a procedural knowledge base of rules, heuristics and processes. Design and engineering are an integral part of tertiary computing education, and their importance is also acknowledged at school levels.

2.3 Computing is a sort of mathematics

Ever since specialized disciplines started to form in the academia, there has been a tight connection between mathematics and sciences. Similarly, during the disciplinary formation of computing, many argued that mathematics and mathematical logic are central to the field: many of the field's problems and their solutions are mathematical, the computer's basic operational principles can be reduced to mathematical logic, many pioneers of the field are mathematicians and the very word 'computing' refers to mathematical activity. Computing has appropriated parts of mathematics, too. Many mathematical objects like graphs, functions and matrices have become computer science objects that can be taught from a computing viewpoint instead of a mathematics viewpoint.

Computing's early emphasis on mathematics was based on ideals as well as practical needs. To get a foothold in traditional research universities, computing should not have looked like a toolmaker, so emphasizing theory created a desirable image of the discipline. And unlike technical inventions, many of which grew obsolete within a decade, theoretical research was

considered to have lasting value. Until the late 1980s, there was a strong narrative of computing being primarily a mathematical field, but a gap was growing between that narrative and what went on in computing education, business and the industry. The software industry struggled with a multifaceted crisis with software production, and theory-oriented research was unable to solve that crisis.

Since the 1990s, debates about the role of mathematics in computing lost their zeal, and today there is a broad consensus on the importance of different kinds of approaches to computing. In universities, the mathematics requirements in computing commonly include discrete mathematics, probability and statistics. Yet still, debates continue about the relationship between computing and mathematics and how much mathematics should a computing professional or a researcher learn. Because in computing, theory and practice are very closely connected, some pioneers emphasize their interplay. Knuth (1991), for example, advised those who spend their time mostly on theory of computing to turn their attention to practice: 'It will improve your theories.' Similarly, he advised those who spend their time mostly on practice to turn their attention to theory: 'It will improve your practice.'

Proponents: Logic rules

Those who argued that much of work in computing actually boils down to mathematics and logic justified their argument by referring to the aims and central questions, necessary skill set, and methods and practices of computing as well as the computer's logical organization. Many aspects of computers and programs can be described by mathematical functions and symbol manipulation. Program states are often practically infinite, and mathematics is the best tool for dealing with infinity. Theory of computing is very much a mathematical theory. Those who advocate stronger inclusion of mathematics in computing education point out that most areas of computing have a close relationship with specific areas of mathematics (Baldwin, Walker & Henderson, 2013).

Methods and practices

Many proponents of a mathematical view of computing argue that one can reason about algorithms, programs and procedures in the same way mathematicians work with functions, theorems and proofs – in their minds or with a pen and paper (Smith, 1998). Hence, for reasoning about programs, skills and knowledge of methods and practices in mathematics and logic are absolutely essential. Even more, formal methods provide a variety of ways for greatly increasing the reliability and robustness of programs.

Even when computing professionals do not use mathematics and logic explicitly, they are a part of our 'computational thinking' habits (Aho, 2011). Although most software engineers, programmers and other computing professionals cope well in their professions without explicitly applying mathematics, mathematics and logic are implicit in very large parts of their work. Consequently, many share the opinion that computing education should include more mathematics, especially more applicable and relevant mathematics (Baldwin, Walker & Henderson, 2013).

Subjects and questions

From the mathematical point of view, computer programs and other objects of computing are essentially abstract objects or can be modelled as such. For example, every executable program can be expressed as an algorithm, which is an abstract object one can reason about in terms of mathematics and logic. Mathematical knowledge is of lasting nature because it consists of necessary truths – truths that cannot be otherwise in the set of axioms and rules where they are stated.

Favourite descriptions of computing's subject matter on the theoretical side are algorithms, classes of computations, models of computing, procedures and abstraction. Yet, there are differences between computing and traditional mathematics: Knuth (1974) wrote that unlike mathematics, computing often deals with finitary constructions and dynamic relationships. While mathematics is more declarative, computing is more imperative, which can be clearly seen in computing's problems, aims and methods. Still, mathematics is the best tool for answering questions about the theoretical limits of different models of computing – 'what can be automated?' – as well as the practical limits set by how much computing is needed – 'what can be efficiently automated?'

Examples: Mathematics and logic in school computing education

In school curricula, mathematical principles include activities like learning how to count from zero to 1,023 using ten fingers for binary digits, learning to use logic to solve mystery problems and learning binary data error detection through card flip magic trick. In the school curriculum recommendations, children learn simple concepts like binary representation in grades K–3 and continue to connections between mathematics, logic and computing in grades 6–9.

Opponents: It's not what we do most of the time

No serious argument has claimed that mathematics plays no role in computing. Instead, the critics argue that mathematical methods do not play a dominant role in computing, that mathematical knowledge and skills are not central to most work in computing and that what matters the most in computing is the computer and its effects on science, society and our lives. Many researchers as well as professional practitioners of computing engage most of the time in activities that are not mathematical by nature: they elicit requirements, design interfaces, build models, debug programs, test systems and write manuals, for example.

Even those who acknowledge computing's mathematical nature often oppose too strong conclusions about the relationship between computing and mathematics. Although both might be dealing with symbols on some level, their aims and objectives differ: mathematics is interested in the relationships between symbols as well as semantics of symbols, while most of computing is

about applications of mathematics to solve problems. And while mathematics is supposed to be independent of any social or human concerns, computing is very much about social and human concerns. One can argue that if computing were only a theoretical discipline, it would have never revolutionized science and the society. It is the machine that counts.

Key concept: Computing as mathematics and logic

Computers are logic machines, some of the most impressive achievements in computing are proven and presented in the language of mathematics and many mathematical structures like matrices, vectors and graphs have become standard computing concepts. Large parts of computing study formal objects, like algorithms and models of computing. Many questions of computing are best solved using mathematics and logic as tools. School curriculum recommendations have acknowledged the tight connection between computational thinking and mathematical as well as logical thinking.

2.4 Computing is a science

Throughout the history of computing, computing and science have been deeply intertwined, and the relationship goes both ways. Firstly, from Newton's prolific numerical calculations to large-scale tabulation operations, computing or numerical analysis has served as a tool for science. Secondly, many pioneers of computing aimed to found computer science on similar scientific principles as natural sciences. Thirdly, many people argue that modern computing started a whole new era of science. Natural sciences gradually developed computational branches – such as bioinformatics, computational physics and computational chemistry – and computer simulation became a central element of progress in science.

The science discussions manifest in many ways. One of them is the debate about the field's name, such as whether computer *science* is the right name. For instance, informatics, algorithmics and datalogy are in use in different countries. Another is concerned with the subject matter of the field: if astronomy is the study of celestial objects and zoology the study of animals, what is computing the study *of*? Yet another branch of the science discussion is concerned with methodology: research fields are characterized by their methods, but what *is* the method of computing – if there is one? And finally, some debates are concerned not with whether computing is a science, but with whether it might be the most important science today, given the amazing success of computational approaches in a broad variety of fields.

One striking feature of computing's science debates is that the debaters do not share a common view of what science is. By science, some mean natural science and some mean any empirical science; some refer to methods and some to theories; and some talk about a body of knowledge or laws and some about a world view. In absence of a common ground, debaters often talk past each other and have great difficulties achieving consensus.

Proponents: It's a new kind of science

Those who argue that computing is a scientific discipline often refer to the aims of science – exploration, description, prediction and explanation of phenomena – and argue that computing shares those aims. Some research in computing also follows the experiment cycle of observation, description, prediction and testing; that cycle is sometimes called the scientific method. Many debaters see computing as an interdisciplinary field that combines theories from a variety of domains and contributes to an even broader variety of fields. People have attempted to resolve the problems with computing's unique scientific nature by describing it as an unnatural science, artificial science, synthetic science and even a completely new domain of science that follows its own paradigm (Rosenbloom, 2013).

Subjects

Many arguments about computing's scientific nature rely on its subject matter. There is a dizzying number of arguments about what computing is a science *of*, ranging from data, information and symbols to algorithms, processes, procedures and information flows as well as complexity, representation, users and designs (Tedre, 2014). All those can be studied through scientific principles. While earlier it was widely agreed that computing's subject matter is artificial – making computing a science of the artificial (Simon, 1969) – newer descriptions of computing argue that computing studies phenomena both artificial and natural. Some have argued that computing has become a 'fourth great domain of science' (Rosenbloom, 2013).

Methods and practices

Computing's large range of subjects makes it a methodologically rich field. Its scope and aims are so broad that it excludes few methodological strategies. Due to its broad variety of subjects of study, such as computability, usability, reliability and efficiency, it flexibly adopts methods from natural sciences and formal fields to social sciences and humanities. But more importantly, computing's amazing success in triggering methodological changes in other fields of science and creating new fields altogether – fields such as bioinformatics, computational physics and computational chemistry – has made many people argue that computing has become the most important of all sciences.

After several decades of increased use of computers in science, the 1980s saw a rapid increase of use of computers to simulate a growing number of phenomena. One by one, research fields spawned 'computational' branches and computer simulation quickly became a central tool for sciences. Today, computing has been characterized as a third pillar of science, alongside the two traditional pillars of theory and experiment. The changes in how research is done using simulations are so radical that those changes have been described as the most disruptive shift since quantum mechanics and the computing era of science as the age of computer simulation (Tedre, 2014).

Examples: Science in school computing education

In many countries, computing is integrated in the teaching of other subject matters. For instance, in grades 9–10, computer simulation is used in physics and biology to represent and experiment on natural phenomena. In the CSTA school curriculum, computational modelling is presented for understanding how interactions between individual elements in complex systems (such as people, animals or cars) give rise to emergent patterns that can be fundamentally unpredictable.

Opponents: If computing is science, it is bad science

Opposition towards computing's status as a science comes in many forms, of which two are particularly common. Firstly, some people argue that computing is not really a science based on a variety of arguments. If by 'science' one means 'natural science', then computing surely is not similar to, for instance, physics and chemistry, because many of its subjects are artificial and because it is not methodologically united. Some also argue that while in the natural sciences, theories compete with each other in explaining the fundamental nature of their subject matters, computing does not have a track record of competing theories of the fundamental nature of, for example, data or information. Neither are theories in computing developed to reconcile theory with anomalies revealed by experiments, which is common in other sciences (Hartmanis, 1994).

Secondly, some people argue that computing is bad science. Large surveys of computing research have revealed that experimental validation of results is not as common in computing as it is in most other sciences. Even further, similar surveys show that research reports in computing often exclude important details of methodology, making it impossible for others to replicate or even properly evaluate the merits of those reports. The role of experiments in computing is another common target of criticism: while natural sciences are driven by crucial experiments, many parts of computing are driven by crucial demonstrations.

Computing's inclusive pick-and-mix attitude towards methodology has been interpreted in two ways. While the proponents of computing's scientific nature see computing's methodological multiperspectivism in a positive light, those who criticize computing's scientific nature quote the same characteristic as methodological eclecticism and lack of shared principles. For many people, 'everything goes' is a sign of methodological anarchism and not a desirable feature of science.

Key concept: Computing as a science

Computers have become the most common tool of science, and simulation has become a standard feature of modern natural science. Many people argue that computing has become a third pillar of science, aside the traditional theory and

experiment. Computing helps theoretically oriented scientists to solve their equations and experimenters to analyse massive amounts of data, and it has given rise to a new way of doing science in the form of simulation.

2.5 Understanding intellectual traditions is important in computing education

The above sections present various windows to computing as a discipline, rooted in the traditions of engineering, mathematics and empirical sciences. They all have their advantages and disadvantages, and all are justified in different ways. As integral parts of computing's practice and theory, the traditions are deeply intertwined and support each other. Some of the greatest achievements of computing happen at the intersections of different intellectual traditions,[1] and others purely within one.

However, combining theory, design and empirical research in one educational program is not always easy. One has to be aware of the limitations of each tradition. For example, one cannot formally prove that a design has the intended qualities or that a computer system will not fail. Showing that something can be built does not demonstrate any of its qualities, such as usefulness, usability or reliability. Empirical research is not a tool for proving things. Explanations of human behaviour are very different from explanations of electromagnetic phenomena. When working in the intersection of computing's traditions, one should know each of them well or risk having results that are flawed from the point of view of each of the traditions.

Working in the intersection of many intellectual traditions has posed problems for educators throughout computing's disciplinary history. Teaching design and engineering requires different educational strategies than teaching the theory of computing. The aims and goals of engineering are different from the aims and goals of theoretical or empirical fields. In theses and degrees on systems and software, engineering ingenuity or programming virtuosity is often not enough, but questions are raised about scientific validation. These issues, and many others, may be alleviated by understanding computing's unique disciplinary ways of thinking and practising and the intellectual traditions behind them. Over the history of modern computing, the field's development has also frequently shifted the focus of computing research and education between different traditions.

Understanding computing's traditions is also important for locating oneself within the landscape of computing fields (see also Chapter 3 in this book). Those who have strong preferences and opinions about computing's essential features will benefit from understanding alternative viewpoints, their strengths and weaknesses. Those who are more ambivalent about their standing will benefit from contemplating different intellectual traditions with regard to courses, curricula and aims of education. A rich and balanced view of computing is also important for students who are still building their identities as computing professionals. By knowing the pros and cons, promises and challenges as well as landmark achievements of each of the major traditions in computing, we can give students a fascinating tour of computing in its full richness.

Key points

- Over the disciplinary history of computing, there has been a lively debate on the field's fundamental nature, aims, methods, essential skills and knowledge, its relations to other disciplines or even the name of the field.
- This chapter introduces three dominant traditions of computing and characterizes their aims, questions, views of subject matter, methods and practices. The traditions are deeply intertwined and support each other.
- Computing as engineering: Many computing activities are centred around requirement analysis, design, implementation, evaluation and production of useful artefacts.
- Computing as mathematics and logic: Computers are logic machines; some of the most impressive achievements in computing are proven and presented in the language of mathematics. Large parts of computing study formal objects, like algorithms and models of computing.
- Computing as a science: Computers have become the most common tool of science, and simulation has become a standard feature of modern natural science. Many people argue that computing has become a third pillar of science, aside the traditional theory and experiment

For further reflection

- Which of computing's design, logico-mathematical and scientific aspects are you most familiar with? What are the most motivating assignments for students in each tradition? Which school subjects do students consider to be closest to computing?
- What kinds of skills and knowledge seem to be important for learning computing principles? How can computing be integrated into other subjects? What emphasis do you place on the three traditions discussed in this section when considering your students' abilities in computing?

Note

1 In this context, the term 'intellectual' is being used to mean discipline-based or scholarly traditions which computing historically draws on.

References

Aho, A. V. (2011), 'Ubiquity Symposium: Computation and Computational Thinking', *Ubiquity*, 2011 (January). doi: 10.1145/1895419.1922682.

Baldwin, D., Walker, H. M., and Henderson, P. B. (2013), 'The Roles of Mathematics in Computer Science', *ACM Inroads*, 4 (4): 74–80.

Brooks, F. P., Jr. (1996), 'The Computer Scientist as Toolsmith II', *Communications of the ACM*, 39 (3): 61–8.

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. (1989), 'Computing as a Discipline', *Communications of the ACM*, 32 (1): 9–23.

Hartmanis, J. (1994), 'Turing Award Lecture on Computational Complexity and the Nature of Computer Science', *Communications of the ACM*, 37 (10): 37–43.

Knuth, D. E. (1974), 'Computer Science and Its Relation to Mathematics', *American Mathematical Monthly*, 81 (April): 323–43.

Knuth, D. E. (1991), 'Theory and Practice', *Theoretical Computer Science*, 90 (1): 1–15.

Rosenbloom, P. S. (2013), *On Computing: The Fourth Great Scientific Domain*, Cambridge, MA: MIT Press.

Simon, H. A. (1969), *The Sciences of the Artificial*, Cambridge, MA: MIT Press.

Smith, B. C. (1998), *On the Origin of Objects*, Cambridge, MA: MIT Press.

Tedre, M. (2014), *The Science of Computing: Shaping a Discipline*, New York: CRC Press/Taylor & Francis.

Wegner, P. (1970), 'Three Computer Cultures: Computer Technology, Computer Mathematics, and Computer Science', in F. L. Alt and M. Rubinoff (eds), *Advances in Computers*, vol. 10, 7–78, Cambridge, MA: Academic Press.

Wegner, P. (1976), 'Research Paradigms in Computer Science', in *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, 322–30, Los Alamitos, CA: IEEE Computer Society Press.

3

Perspectives on Computing Curricula

Erik Barendsen and Mara Saeli

Chapter outline

Chapter synopsis

In this chapter, we take a look at computer science curricula from various perspectives. We present the theories behind these perspectives as well as characteristic examples taken from actual curriculum documents and practices. First, we explore the various components of a curriculum, in particular the underlying rationale, the content matter and the goals and objectives. Second, we present a way to view curricula in terms of implementation layers, each with its own level of abstraction, and identify influencing factors, including the teachers' own expertise and beliefs. We highlight the special role of textbooks and other teaching materials as an intermediate layer between designed and implemented curricula. We argue how the perspectives presented in this chapter can be used as lenses for analysing and understanding existing curricula as well as for supporting curricular decision-making.

3.1 Introduction

Understanding curricula

What is taught in computer science in schools ('the curriculum') is influenced by many factors. It depends, for example, on government policies and national curricula, standards and guidelines, textbooks and local school policies. Finally, teachers' personal expertise, preferences and beliefs shape the contents of their lessons.

This chapter is meant to help educators to make sense of these factors and their respective influence. It provides them with lenses to analyse and understand curricula, to participate in discussions about computer science education and to reflect on their own teaching practice from a curricular point of view. Thus, we aim to contribute to the readers' 'curricular literacy' with the purpose of supporting informed decision-making about content matter.

The chapter takes a view on curricula from two perspectives. First, we discuss curricula in terms of their constituting *components*. Second, it will be helpful to distinguish *layers* in which curricula are represented, ranging from curricular intensions to implementations in classroom practice.

Curriculum components

Overall, a curriculum gives information on various aspects of teaching and learning. Thijs and van den Akker (2009) call them *components* (see Table 3.1). The rationale is the most fundamental component, influencing many others.

The *curricular spider web* (see Figure 3.1) outlines the different aspects described in a curriculum – if one changes one aspect, the others will be affected by this change. The centre of the curricular spider web is the rationale, answering the core question why students should learn. The interconnectedness of the components stresses that the components should be coherent ('aligned') to be effective.

Table 3.1 Curriculum components

Component	Core Question
Rationale	Why are they learning?
Aims and objectives	Towards which goals are they learning?
Content	What are they learning?
Learning activities	How are they learning?
Teacher role	How is the teacher facilitating their learning?
Materials and resources	With what are they learning?
Grouping	With whom are they learning?
Location	Where are they learning?
Time	When are they learning?
Assessment	How is their learning assessed?

Source: Thijs and van den Akker (2009:12).

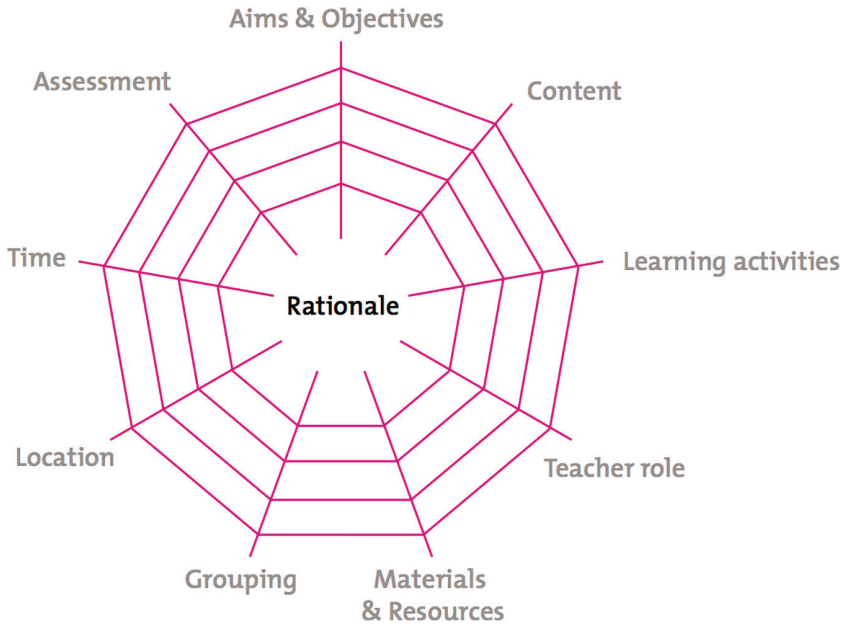

Figure 3.1 The curricular spiderweb (*Source:* Thijs and van den Akker, 2009:11)

Thijs and van den Akker describe the curricular spider web as a fragile thing:

> Although a spider web is relatively flexible, it will most certainly rip if certain threads are pulled at more strongly or more frequently than others. The spider web thus illustrates a familiar expression: every chain is as strong as its weakest link. It may not be surprising, therefore, that sustainable curriculum innovation is often extremely difficult to realize. (Thijs and van den Akker, 2009: 12)

As a consequence, the underlying rationale that holds together the parts and provides coherence to the curriculum can be identified (see Table 3.1).

Moreover, in order to use a curriculum document to review and refine standards and plan teaching, it is crucial to be aware of the central component of the curricular spider web, the underlying rationale. By doing so, the wording, choices and gaps in a curriculum can be understood and explained.

Curriculum layers

In addition, to understand the effects of a curriculum, it is useful to distinguish between the curriculum as it was conceived, on the one hand, and the teaching practice, on the other. Van den Akker (2004: 3) developed a framework describing the different types of curriculum representations:

- Intended curriculum:
  - *Ideal*: The vision of the society underlying the curriculum
  - *Formal/written*: Documents describing a national curriculum; specifies intentions

Table 3.2 Curricular levels and influence factors

Curriculum Model (van den Akker, 2004: 3)			Influenced by …
Intended	Ideal	Vision (rationale or basic philosophy underlying the curriculum)	Perspectives on higher level: policies, standards (*actors*: policymakers, standards, government, experts, professional organizations like ACM, CAS, GI, etc.) … shared/collective perspectives
	Formal/written	Intentions	
Implemented	Perceived	Interpretation by users	Influenced by teachers' perspectives on computing (Tedre); influenced by curriculum emphasis (van Driel) Influenced by teachers' orientations, knowledge (PCK, CK), beliefs *Actors*: teachers … individual perspectives
	Operational	Actual process of teaching/ learning	
Attained	Experiential	As experienced by learners	Learners' beliefs, interests, motivation etc. … individual perspectives
	Learned	Learning outcomes	

- Implemented curriculum:
  - *Perceived*: How teachers perceive and interpret the curriculum
  - *Operational*: The so-called curriculum-in-action – what is taught and learned in the classroom
- Attained curriculum:
  - *Experiential*: How learners experience their learning
  - *Learned*: The learning outcomes of the learners

The intended curriculum is often described as a document on a national level or is presented as a suggestion from a teacher association, aiming to influence what is taught in schools. These usually (or hopefully) rely on a rationale that is shared by the community. On a more local level, such a curricular framework is then implemented at schools and in individual classrooms – this may be done by a group of teachers for a local school or by an individual teacher outlining a curriculum (lesson plan) for the next school year. How this is done depends on individual perceptions and interpretations. One final aspect is the level of the attained curriculum, capturing the learners' view on the implemented curriculum – it describes the learning results.

Table 3.2 presents some influencing factors and the curricular levels they influence. It summarizes some concepts and theories, some of which we will present and discuss in the following sections.

Textbooks and other organized resource materials (hereafter called 'textbooks' for short) often play a mediating role between the intended and the implemented curriculum. Valverde et al. (2002) call this intermediary level the *potentially implemented curriculum* (see Figure 3.2). Although there is no international empirical evidence of the use of textbooks in computer science education, we know from a survey that more than 60 per cent of Dutch teachers use standard textbooks, while almost 40 per cent use other resources or personally developed materials

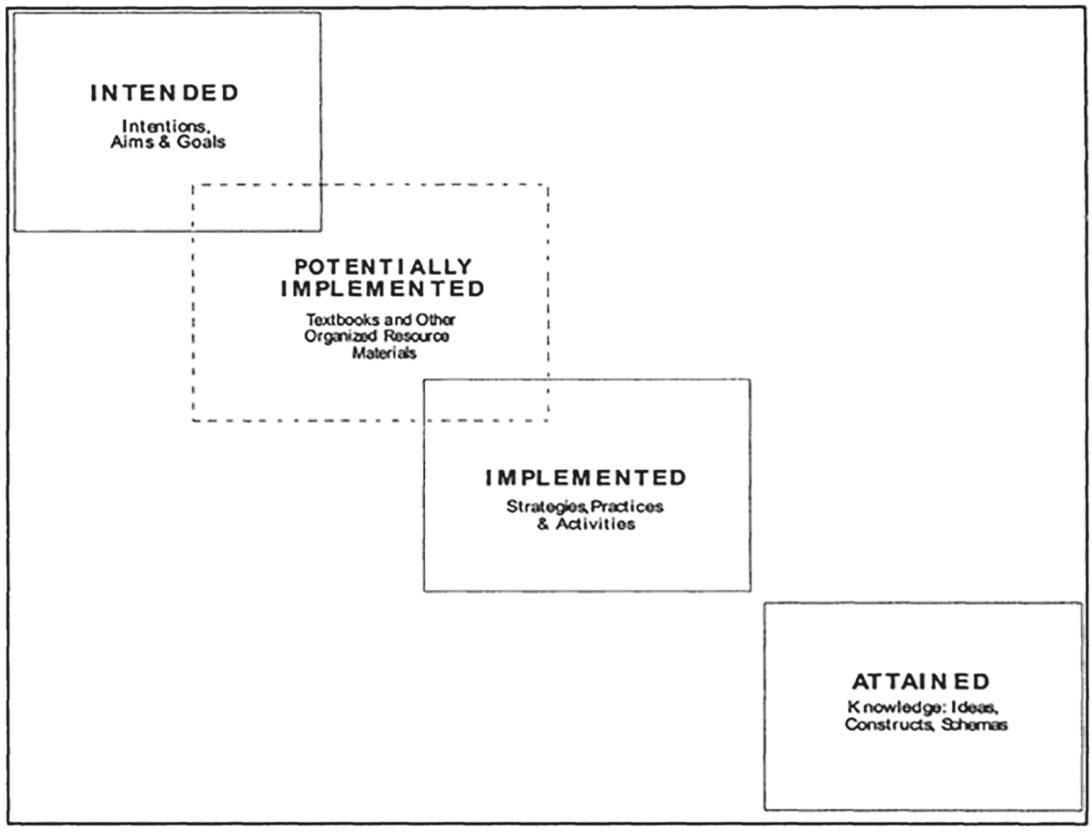

Figure 3.2 The mediating role of textbooks (*Source:* Valverde et al., 2002)

(Tolboom, Krüger and Grgurina, 2014). Textbook authors make specific decisions regarding the most appropriate sequencing of the content and the structuring of pedagogical situations where activities, explanations, examples and exercises are assigned particular roles (Valverde et al., 2002)

Structure of this chapter

In the remaining of this chapter, we will explore three of the curriculum components and show how they can be used to understand, compare and make sense of curricular representations. Then, we will look into two different curriculum layers, giving readers a few opportunities to reflect on their practice and which factors influence it. We will look first at the potentially implemented curriculum (e.g. textbooks and teaching material) and finally at the implemented curriculum (e.g. teaching practices and strategies). All this comes together in the closing section, in which we will suggest how to apply the presented perspectives to reflect on teachers' curricular decision-making.

3.2 Curriculum components

Rationale

The underlying rationale of a curriculum is crucial for alignment of content and internal coherence. In essence, the rationale answers the question: Why teach this? The rationale therefore needs to take general aims of education into account and interpret these in terms of the subject matter of the discipline.

In an analysis of the role of programming, Schulte (2013) identifies the following general goals:

- *Cope with affordances*: developing competencies and skills to be able to cope with affordances of life in a variety of different situations.
- *Participation in society/democracy*: being able to act in terms of participation. It refers to what is called 'education for democratic citizenship'. This is the ability to engage with and responsibly participate in the democratic sustainment and development of society.
- *Development of identity*: developing a sense of identity; getting to know what interests me, what resonates with me, what repels me, and so on.

These goals reflect the relative consensus on the general rationale of education, articulated by Tyler (1949): to provide knowledge, social preparation and personal development.

The same elements are elaborated upon by Biesta (2015), who distinguishes between the following:

- *Qualification*: gaining knowledge, skills, dispositions, and so on needed to go on and do something, either specific, such as a profession, or in general, such as being qualified to live in a complex society
- *Socialization*: becoming part of society (existing ways for doing and being) – culture and tradition
- *Subjectification*: becoming more autonomous and independent in thinking and acting

The question for a curricular rationale can thus be rephrased as follows: 'What is the role of the subject domain in these general goals of education?' The perceived nature of the discipline (see Chapter 2) is an important influencing factor for such a general vision of computer science education.

In the introduction of the CAS curriculum document (CAS, 2012), each of the three general goals is addressed: *'Pupils studying computing gain insight into computational systems of all kinds, whether or not they include computers'* (qualification); *'Education enhances pupils' lives as well as their life skills. It prepares young people for a world that doesn't yet exist, involving technologies that have not yet been invented, and that present technical and ethical challenges of which we are not yet aware'* (socialization) and, less prominently, *'This … makes it an extraordinarily useful and an intensely creative subject, suffused with excitement, both visceral* ("it works!") *and intellectual ("that is so beautiful")'* (subjectification).

The CSTA (Computer Science Teachers Association) standards contain elements of socialization and a more prominent subjectification rationale:

> K–12 computer science teachers can thus nurture students' interests, passions, and sense of engagement with the world around them by offering opportunities for solving computational problems relevant to their own life experiences.
>
> (CSTA, 2016: 5)

Many educators regard the competence to *express oneself* as important – or even a prerequisite. This self-expression side of subjectification is more prominent in arts-related subjects. If one looks closely, however, this aspect can also be found in computing.

Besides its role in discovering new curriculum elements, awareness of the underlying rationale is crucial for maintaining the coherence and balance of the components constituting the curricular spiderweb.

Example: Storytelling in programming

Storytelling is sometimes used in introductory programming (e.g. producing small animations in Scratch by primary school students). Doing so, the students indeed learn about programming – but the animation itself does not have a particular usefulness; it does not solve a problem. Hence, the value of these tasks is sometimes seen only in its contribution to learning problem-solving techniques. However, based on the general goal of subjectification, the ability to write an animation as a self-expression has value in its own right. Indeed, it could be possible to extend the role of programming to 'expression': the idea that learners learn to program as a means to produce aesthetic expressions (Schulte, 2013) and hence as a contribution to subjectification as the third general goal of education.

Content

In this subsection, we will look at content matter in curricula in terms of computer science *concepts*. The term 'concepts' refers to 'topics and ideas belonging to the subject matter, regardless of the specific skills or attitudes in which they appear' (Barendsen et al., 2015: 85).

A classification of concepts in curricula can be an effective way to compare curricula on a global level. The underlying idea is that the occurrence of concepts gives an idea of the content emphasis in the documents. Moreover, the number of occurrences in curriculum documents can be regarded as a (global) indicator of the relative importance of concepts and knowledge categories. Such a concept analysis is relatively easy to carry out and results in a useful overview of the specified content matter. This concept distribution can serve as a starting point for a more in-depth comparison of curricula. In this way, Barendsen and Steenvoorden (2016) compared the curricula of CSTA and CAS as well as the national curriculum descriptions of France and the Netherlands (before and after the curriculum revision of 2016).

Example: Concept analysis of curriculum documents

Barendsen et al. (2015) and Barendsen and Steenvoorden (2016) apply a content analysis method to various curriculum documents. Concepts were extracted from

curriculum texts and grouped into general categories based on the list of so-called knowledge areas of the ACM/IEEE-CS Joint Task Force on Computing Curricula (2013): algorithms, architecture, modelling, data, engineering, intelligence, mathematics, networking, programming, security, society, usability and a final rest category.

Figure 3.3 displays the relative distribution of concept occurrences.

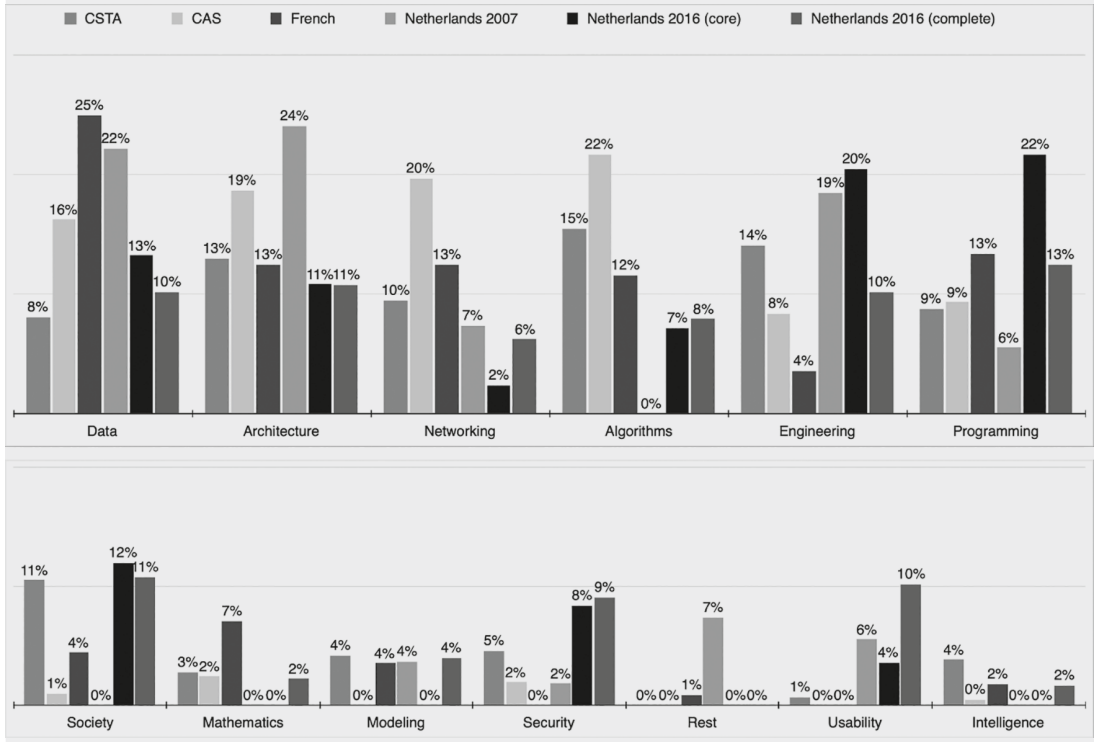

Figure 3.3 Relative distribution of concept categories (*Source:* Barendsen and Steenvoorden, 2016)

An interesting way to compare curricula is to look at the most prominent categories; see Table 3.3.

Table 3.3 Top 5 knowledge categories in the sample curricula*

CSTA	CAS	France	Netherlands 2007	Netherlands 2016
Algorithms	Algorithms	Data	Architecture	Programming
Engineering	Networking	Programming	Data	Engineering
Architecture	Architecture	Architecture	Engineering	Data
Society	Data	Networking	Networking	Society
Networking	Programming	Algorithms	Rest	Architecture

* Most frequent categories are mentioned first.

Using the framework of computing traditions (Chapter 2), we can try to explain some differences. The CSTA and Dutch documents emphasize the engineering ('making') aspect of computing, which accounts for the relatively high frequencies of engineering and programming concepts. Also, from a more in-depth content analysis, it appears that in the CAS curriculum and French document, the scientific tradition is more present (and in the French case, the mathematical tradition).

Conceptual choices, priorities and applications can often be explained using Tedre's framework of computing traditions (see Chapter 2). Let us, for example, speculate about typical roles of the concept of the Turing Machine in the respective traditions. As an important analytic model, the Turing Machine would be a primary concept in any curriculum rooted in the mathematical tradition. In the engineering tradition, however, one could expect a more marginal role of the Turing Machine as a theoretical basis of more prominently visible principles of (non-)computability. In the science tradition, the Turing Machine can be expected to serve as a descriptive model for the way computers work.

Aims and objectives

Curriculum descriptions usually contain learning objectives in terms of knowledge, skills and attitudes, as opposed to just a specification of content matter. Such a description specifies the aims of the curriculum in terms of students' outcomes. To prepare for a proper alignment of the learning objectives with assessment, learning objectives are usually formulated in operational terms (i.e. in terms of observable student behaviour).

Various taxonomies exist for facilitating the formulation of operational objectives. One of the best known of those is the revised Bloom's taxonomy (Anderson and Krathwohl, 2001) classification:

- Remember (characteristic verbs: recognizing, recalling)
- Understand (explaining, summarizing)
- Apply (show, solve)
- Analyse (categorize, subdivide)
- Evaluate (construct, design)
- Create (criticize, recommend)

It has been argued that this taxonomy does not fit seamlessly into computer science, because the operational objectives within computing do not seem to be equally distributed across the levels of this taxonomy. Indeed, many 'making' activities will be categorized in the higher Bloom categories (e.g. Lister and Leany, 2003), for example, 'create'. In this respect, Johnson and Fuller (2006) propose to add a level 'higher application'.

Another possible refinement is provided by the SOLO taxonomy (Biggs and Collis, 1982), distinguishing levels according to the 'scope' of students' skills, varying from a local to a holistic perspective, with the following key aspects:

- Pre-structural: unconnected, unorganized
- Unistructural: local perspective
- Multistructural: multi-point perspective
- Relational: holistic perspective
- Extended abstract: generalization and transfer

Many authors use complexity by referring to local versus holistic perspectives ('scopes') (e.g. lines in a program versus interactions in a multicomponent system) (see Lister et al., 2006; Whalley et al., 2006). However, if used synergically, the two taxonomies can help navigate through the aims and objectives of a curriculum. For example, in curriculum specifications, we can find indications of 'lower' Bloom's taxonomy level being achieved by the accomplishment of other 'higher' levels.

Example: Using Bloom's Taxonomy

The ability to *understand* and *explain* a program is much more important than the ability to produce working but incomprehensible code. Depending on the level, pupils should be able to do the following (CAS, 2012):

- *Design* and write programs that include
  - *Sequencing*: doing one step after another
  - *Selection* (if-then-else): doing either one thing or another
  - *Repetition*: iterative loops or recursion
  - *Language constructs that support abstraction*: wrapping up a computation in a named abstraction, so that it can be reused (the most common form of abstraction is the notion of a 'procedure' or 'function' with parameters)
  - Some form of interaction with the program's environment, such as input/output, or event-based programming
- Find and correct errors in their code
- Reflect thoughtfully on their program, including assessing its correctness and fitness for purpose, understanding its efficiency and describing the system to others

In this example, a lower Bloom level (understand and explain) is realized by achieving other higher levels (design, write and reflect). During this process of interaction between the different Bloom levels, one might expect the students to reach different levels of SOLO complexities as they move back and forth from the 'making/creating' activities.

The rather central role of the 'making' aspect of computer science is also addressed in the explanation given by the Dutch national curriculum committee.

Informatics is seen by many as a constructive discipline: a subject area where creating things (mostly digital artefacts) is the key element. In this epistemic view, informatics as a scientific discipline supplies the conceptual and procedural knowledge about such artefacts and the creation process. The 'creation' perspective is an attractive starting point for the subject. Therefore, the

committee has decided to position 'design and development' as a central skill in the new curriculum (Barendsen, Grgurina and Tolboom, 2016: 109).

One can recognize a strong 'engineering' emphasis, with maths/science as supporting traditions.

In summary, formulating goals and objectives for a given curriculum content is not straightforward. First, taxonomies are useful tools to be precise about the intended outcome levels. In the case of computer science, it is fruitful to complement a purely cognitive framework like Bloom's with domain-specific complexity indicators in SOLO terms. Second, the underlying perception of the nature of the discipline impacts the translation of content into specific learning goals.

3.3 Curriculum layers

So far, we have discussed the intended curriculum through the perspective of the different components (rationale, content, aims and objectives). These components can be used as lenses for analysing and understanding existing curricula. In this section, we will focus on the perspective of curriculum layers (potentially implemented and implemented) to explore how the intended curriculum is translated into textbooks and classroom practice.

The potentially implemented curriculum

Textbook authors 'translate' the intended curriculum into documents including a sequence of topics, tasks and performance expectations which then teachers might use in their practice. During this translation process, textbook authors provide a first interpretation of the standards prescribed in the intended curriculum. A textbook can be seen as a tangible view of the intended curriculum, through the lenses of the textbook authors' rationale.

The development of textbooks can be regulated by authorities, but in some cases (e.g. in the Netherlands), there is no authority which recommends, certifies or approves textbook series before they are distributed on the market. This could, for example, lead to textbooks series offering different interpretations of the national intended curriculum. Van Zanten and van den Heuvel-Panhuizen (2014) conducted a textbook analysis in the context of mathematics education, in which they examined how two different Dutch textbook series differ in their view on subtraction. The results of their analysis show that the investigated textbook series vary in their agreement with the intended curriculum with respect to content and performance expectations. The textbook series reflect divergent views on subtraction up to 100 as a mathematical topic. Consequently, the examined textbook series provide very different opportunities to students to learn subtraction up to 100.

Example: Textbook analysis activity for teachers

In their textbook analysis, Van Zanten and van den Heuvel-Panhuizen (2014) examined two textbook series. The analysis focused on three perspectives: the

subject content, the performance expectations and the learning facilitators. It is likely that the teaching material of your choice (e.g. textbook series and online material), unless you have created your own, provides a redefined intended curriculum, which you will then further interpret and implement. Compare one topic of your choice from the teaching material against the curriculum standards. How does the potentially implemented redefinition of that topic differ from yours, in terms of the content, the performance expectations and the learning facilitators? Is the teaching material aligned with your rationale? And consequently, how do the other components of the curricular spiderweb compare with your interpreted/perceived curriculum?

The implemented curriculum

In the remainder of this section, we will focus on the implemented curriculum: the ways teachers perceive and put the intended curriculum into action. This process of implementing the curriculum is influenced by a range of teacher factors, like knowledge such as content knowledge (CK) and pedagogical content knowledge (PCK), orientations and beliefs (Shulman, 1986).

The space for teachers to give a personal interpretation in implementing a formal curriculum also became apparent in a PCK study by Rahimi, Barendsen and Henze (2016) in the context of the Dutch formal curriculum, which stresses the engineering perspective as we have seen. In the teachers' PCK, Rahimi, Barendsen and Henze found a broad range of goals and objectives connected to the topic 'design and development', ranging from purely conceptual objectives to more practical learning goals. For computer science education, several studies found a relationship between beliefs and teaching (e.g. Fessakis and Karakiza, 2011; Schulte and Bennedsen, 2006; Bender et al., 2016).

Teachers' *curriculum emphasis* is considered to be one of these beliefs. The idea is that teachers have goals that lie beyond the subject itself; such an emphasis has also the role to send a message about the subject. In a study with Dutch chemistry teachers, van Driel, Bulte and Verloop (2007, 2008) found three different curriculum emphases: fundamental chemistry (FC), chemistry, technology and society (CTS) and knowledge development in chemistry (KDC). FC corresponds to the belief that theoretical notions should be taught first, because such notions can provide a basis for subsequently understanding the natural world and are needed for the students' future education. CTS implies an explicit role of technological and societal issues within the chemistry curriculum. Finally, KDC is connected with the idea that students should learn how knowledge in chemistry is developed in sociocultural contexts, so that they learn to see chemistry as a culturally determined system of knowledge, which is constantly developing (cf. Barendsen and Henze, 2019).

To our knowledge, this framework has not been studied empirically in computer science education, but we can use the curriculum emphasis as a framework to discuss how a teacher makes decisions when implementing a curriculum. We can speculate about a version of the framework for computer science: FC is then the idea that the fundamental concepts of computer science should be emphasized, CTS is the notion that the interaction between computing and society should be emphasized and KDC stresses the way of thinking like a computer scientist.

For example, three teachers with different curriculum emphases will be teaching the same topic (e.g. computational thinking) using three different approaches. A teacher with a focus on FC probably would stress the need to first teach fundamental concepts like algorithm as a prerequisite for engaging in computational thinking. From a CTS point of view, a teacher might stress computational thinking as a way to have some impact on society and technology. While a teacher with focus on KDC would probably aim to teach learners to think like a computer scientist. Van Driel, Bulte and Verloop (2007, 2008) found in their study that most teachers tend to support all three emphases to some degree, their focus being influenced by their experience and beliefs.

These three curriculum emphases seem to be closely connected to the general rationale of education articulated by Biesta (2015) mentioned earlier. From the perspective of the specific subject (e.g. chemistry, computer science, etc.), FC relates to qualification, where gaining knowledge, skills and dispositions is the basis (the knowledge needed to be a computer scientist); CTS and socialization both look at the perspective of the integration between education and society; while KDC and subjectification both address how the specific knowledge and way of thinking within a subject have a role in sociocultural contexts.

A final personal factor is presented by Ni (2011), in terms of computer science *teacher identity*. This comprises 'Perception of CS', 'Perception of Teaching' and 'Perception of Self as a CS Teacher'. These three perceptions interact with the educational background of the teacher, the curriculum and (Ni stresses this point) the availability of a CS teacher community.

We have seen that personal factors influence the implementation of formal curricula when it comes to interpreting and elaborating the higher-level learning goals and when making decisions on how to implement the curriculum. Bishop and Whitfield (1972) provide a framework depicting the elements of this decision-making process, including a chain of factors, from *background and experience* via *beliefs and values* to *aims and objectives*. A so-called *decision schema* connects these (personalized) aims and objectives to the teaching situation, resulting in *decision and action* (see Borko, Roberts and Shavelson, 2008). This influence not only is visible in the long-term (deliberate, advised) planning of teacher and student activities but also plays a role in decision-making 'on-the-spot' (i.e. during teaching). In such moments, all threads of the curricular spiderweb come together.

3.4 Curricular decision-making

In this chapter, we suggest a journey through the curricular spiderweb (Thijs and van den Akker, 2009), which gives prospective and in-service teachers an opportunity for reflection on their own implemented curriculum and provides the tools to make informed decisions. We guide the reader through a visit of the curricular spiderweb, looking at the connections from the different components and layers.

Starting from the central component of the curricular spiderweb, we can reflect on the underlying rationale of a curriculum and the different general goals of education: qualification, socialization and subjectification. Also the perceived nature of the discipline is an important influencing factor for a general vision of computer science education and its implementation in the classroom.

Analysing the content component in terms of concepts and knowledge categories of a curriculum turned out to be a useful approach to uncover some underlying principle beneath the selection of content in curricula. In this case, it was also useful for discussing and discovering various traditions in otherwise seemingly incomparable documents.

The last component from the curricular spiderweb is the aims and objectives, in terms of formulating goals and objectives for a given curriculum content. Looking at an example from the CAS standards (2012), we can examine its aims and objectives using both a cognitive (Bloom) and a domain-specific complexity indicator (SOLO).

After having looked at the different components, we propose to observe a curriculum from two different layers, offering the readers further opportunity for reflection on their practice. The first layer is the potentially implemented curriculum: textbooks and teaching material.

Lastly, the teachers' own beliefs influence their teaching practice and hence the implemented curriculum layer. Being aware of the underlying perspectives, views and emphases in a curriculum and of one's own identity as a teacher can help to reflect on decisions and thus enhance this decision-making and one's own professional identity as a teacher. One aspect of this is the belonging to a community of computer science teachers. As quoted at the beginning of the chapter, curricula in computer science education will continue to evolve. Thus, we believe as a teacher you should not only be aware of implementing the curriculum in your classroom but also be able to participate in refining the formal curriculum.

As mentioned earlier, there are many stakeholders involved in using and refining a curriculum, not least the teachers. This chapter provides prospective and in-service teachers reflection opportunities and tools to actively maintain a connection with the intended curriculum in their practice and take informed *curriculum decisions* when, for example, choosing teaching materials (e.g. textbooks).

Key points

- The term 'curriculum' comprises various components, that is aspects of teaching and learning. If one changes one aspect, the others will be affected. The visualization of components in a 'curricular spiderweb' emphasizes that the components should be coherent ('aligned') and (to some extent) flexible to be effective.
- The underlying rationale of a curriculum is crucial for alignment of content and internal coherence. In essence, the rationale is the answer to the question 'Why learn this?' and thus reinterprets general roles and aims of education in terms of the curriculum content matter, in this case computer science.
- The content of a curriculum can be characterized in terms of concepts and knowledge categories. These provide a way to compare documents that otherwise differ in style and presentation.
- Taxonomies like Bloom's and SOLO can help to formulate aims and objectives in a precise and operational way.
- The intended curriculum is often described as a document on a formal, in practice, national level. On a local level, there is the implemented curriculum: the ways

teachers perceive the intended curriculum and put it into action. This provides space for teachers to give a personal interpretation in implementing a formal curriculum, depending on individual perceptions, beliefs and interpretations. Teachers' curriculum emphasis is considered to be one of these beliefs.
- Textbooks are a mediating role between the intended and the implemented curriculum. Textbook authors provide a first interpretation of the standards prescribed in the intended curriculum.
- The three traditions of computing (see Chapter 2) help explain curricular aims, questions, views of subject matter, methods and practices.

For further reflection

How do you, as a pre- or in-service teacher, see the relationship and impact of computer science to these general goals of education? Which of the computing traditions (engineering, mathematical, scientific) are you most familiar with? How is this reflected in your teaching practice?

Analyse a textbook and compare one topic of your choice from the textbook against the curriculum standards (the intended curriculum). In this process, reflect on how the potentially implemented redefinition of that topic differs from yours, in terms of the content, the performance expectations (intended learning outcomes) and the learning facilitators (scaffolding strategies).

Lastly, reflect on your own teaching beliefs in terms of computer science education on your teaching practice, that is, the implemented curriculum layer. In this process, reflect on your personal curriculum emphasis and your personal identity within computer science.

References

ACM/IEEE-CS Joint Task Force on Computing Curricula (2013), *Computer Science Curricula 2013 (Tech. Rep.)*, ACM Press and IEEE Computer Society Press.

Anderson, L. W., and Krathwohl, D. R. (2001), *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*, Harlow: Longman.

Barendsen, E., Grgurina, N., and Tolboom, J. (2016), 'A New Informatics Curriculum for Secondary Education in the Netherlands', in A. Brodnik and F. Tort (eds), *Informatics in Schools: Improvement of Informatics Knowledge and Perception*, 105–17. New York: Springer.

Barendsen, E., and Henze, I. (2019), 'Relating Teacher PCK and Teacher Practice Using Classroom Observation', *Research in Science Education*, 49, 1141–75. doi: 10.1007/s11165-017-9637-z.

Barendsen, E., Mannila, L., Demo, B., Grgurina, N., Izu, C., Mirolo, C., Sentance, S., Settle, A., and Stupuriene, G. (2015), 'Concepts in K–9 Computer Science Education', in *Proceedings of the 2015 ITiCSE on Working Group Reports*, 85–116, ACM.

Barendsen, E., and Steenvoorden, T. (2016), 'Analyzing Conceptual Content of International Informatics Curricula for Secondary Education', in A. Brodnik and F. Tort (eds), *Informatics in Schools: Improvement of Informatics Knowledge and Perception*, 14–27. Cham: Springer.

Bender, E., Schaper, N., Caspersen, M. E., Margaritis, M., and Hubwieser, P. (2016), 'Identifying and Formulating Teachers' Beliefs and Motivational Orientations for Computer Science Teacher Education', *Studies in Higher Education*, 41 (11): 1–16. doi: 10.1080/03075079.2015.1004233.

Biesta, Gert J. J. (2015), Good Education in an Age of Measurement: Ethics, Politics, Democracy. Abingdon: Routledge.

Biggs, J. B., and Collis, K. F. (1982), *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Cambridge, MA: Academic Press.

Bishop, A. J., and Whitfield, R. C. (1972), *Situations in Teaching*. New York: McGraw Hill.

Borko, H., Roberts, S. A., and Shavelson, R. (2008), 'Teachers' Decision Making: From Alan J. Bishop to Today', in P. Clarkson and N. Presmeg (eds), *Critical Issues in Mathematics Education*, 37–67, Berlin: Springer.

Computing at School (2012), *Computer Science: A Curriculum for Schools*. https://www.computingatschool.org.uk/resource-library/2012/september/computer-science-a-curriculum-for-schools. CAS Working Paper.

CSTA (2016), [INTERIM] CSTA K–12 Computer Science Standards Revised 2016. https://csteachers.org/documents/en-us/17da0b29-1303-4651-889c-8598d5fe2a5d/1/.

Fessakis, G., and Karakiza, T. (2011), 'Pedagogical Beliefs and Attitudes of Computer Science Teachers in Greece', *Themes in Science & Technology Education*, 4 (2): 75–88.

Johnson, C. G., and Fuller, U. (2006), 'Is Bloom's Taxonomy Appropriate for Computer Science?' in *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, 120–3. ACM. http://dl.acm.org/citation.cfm?id=131582.

Lister, R., and Leaney, J. (2003), 'Introductory Programming, Criterion-Referencing, and Bloom', *ACM SIGCSE Bulletin*, 35 (1): 143–7.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., and Prasad, C. (2006), 'Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy', *SIGCSE Bulletin*, 38: 118–22.

Ni, L. (2011), 'Building Professional Identity as Computer Science Teachers: Supporting Secondary Computer Science Teachers through Reflection and Community Building', in *Proceedings of the seventh ICER '11*, 143, ACM Press.

Rahimi, E., Barendsen, E., and Henze, I. (2016), 'Typifying Informatics Teachers' PCK of Designing Digital Artefacts in Dutch Upper Secondary Education', in A. Brodnik and F. Tort (eds), *Informatics in Schools: Improvement of Informatics Knowledge and Perception*, 65–77, Cham: Springer.

Schulte, C. (2013), 'Reflections on the Role of Programming in Primary and Secondary Computing Education', in *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, 17–24, New York: ACM Press.

Schulte, C., and Bennedsen, J. (2006), 'What Do Teachers Teach in Introductory Programming?' in *Proceedings of the Second ICER'06*, 17–28, ACM.

Shulman, L. S. (1986), 'Those Who Understand: Knowledge Growth in Teaching', *Educational Researcher*, 15 (2): 4–14.

Thijs, A., and van den Akker, J. (2009), *Curriculum in Development*, Enschede: SLO.

Tolboom, J., Krüger, J., and Grgurina, N. (2014), *Informatica in de bovenbouw havo/vwo: Naar aantrekkelijk en actueel onderwijs in informatica*, Enschede: SLO.

Tyler, R. W. (1949), *Basic Principles of Curriculum and Instruction*, Chicago, IL: University of Chicago Press.

Valverde, G. A., Bianchi, L. J., Wolfe, R. G., Schmidt, W. H., and Houang, R. T. (2002), *According to the Book: Using TIMSS to Investigate the Translation of Policy into Practice through the World of Textbooks*. Dordrecht: Kluwer Academic.

Van den Akker, J. (2004), 'Curriculum Perspectives: An Introduction', in *Curriculum Landscapes and Trends*, 1–10. Dordrecht: Springer. http://link.springer.com/chapter/10.1007/978-94-017-1205-7_1.

Van Driel, J. H., Bulte, A. M. W., and Verloop, N. (2007), 'The Relationships between Teachers' General Beliefs about Teaching and Learning and Their Domain Specific Curricular Beliefs', *Learning and Instruction*, 17 (2): 156–71.

Van Driel, J. H., Bulte, A. M. W., and Verloop, N. (2008), 'Using the Curriculum Emphasis Concept to Investigate Teachers' Curricular Beliefs in the Context of Educational Reform', *Journal of Curriculum Studies*, 40 (1): 107–22.

Van Zanten, M., and Van den Heuvel-Panhuizen, M. (2014), 'Freedom of Design: The Multiple Faces of Subtraction in Dutch Primary School Textbooks', in Y. Li and G. Lappan (eds), *Mathematics Curriculum in School Education: Advances in Mathematics Education,* Dordrecht: Springer.

Whalley, J., Lister, R., Thompson, E., Clear, T., Robbins, P., and Prasad, C. (2006, 16–19 January), 'An Australasian Study of Reading and Comprehension Skills in Novice Programmers, Using the Bloom and SOLO Taxonomies', in *Proceedings of the Eighth Australasian Conference Computing Education*, 243–52, Hobart, Australia: Australian Computer Society.

4

Computer Science, Interaction and the World: The ARIadne Principle

Carsten Schulte, Felix Winkelnkemper and Lea Budde

Chapter outline

Chapter synopsis

This chapter addresses the implications of how the strong interrelation between digital technology and everyday life and society can be reflected when teaching computer science. The reader will get an overview of how to address these issues as an integral part of teaching computing. Instead of splitting competences into a technology-based computer science opposed to a usage-based information and communication technology (ICT), and instead of separating technological aspects from their impact and relation to the pupils' world, the ARIadne (architecture relevance genesis and interaction) principle integrates several levels of descriptions by focusing on their interrelations. The ARIadne principle can therefore serve as a tool to support the integration of ethical and societal aspects into the already established core aspects of computer science education.

4.1 Introduction

Computer science teachers can tackle their subject from a number of different perspectives which are discussed using varying terminologies. They have to decide whether they have to teach how to use a piece of software or whether they are concerned only with making its internal workings understandable. They must also decide whether they themselves have to cover the influence of information technology on social and individual life or whether they should leave that task for their colleagues. Last but not least, they must make a decision whether they need to explore with their pupils why the digital world got the way it is or whether it does suffice to teach the status quo.

The fact that questions like these do arise suggests different approaches to what computing education might be. These approaches distinguish themselves from each other in their focus which lies either on internal structures of digital technology – how software and hardware work – or on its application – what they are used for and how they are used. They also have different answers to the question, To what extent the interrelations between digital artefacts and society, on the one hand, and individual users, on the other hand, should be addressed in computing courses?

This chapter promotes an approach to computing education which aims to integrate the two areas. It constitutes that, in a computerized world, computer science education can no longer be limited to the role of covering only the architectural aspects of technology, such as how digital artefacts are constructed and how they work. Covering and answering these aspects of technology, of course, do not disappear. However, it has to be integrated with investigations into how and where digital technology surrounds us, how we use it and how it affects almost every aspect of life. Thus, the more formal (mathematical) foundations behind the architectural aspects and its concrete implementations in technology need to be complemented with aspects of their relevance and meaning in relation to social and personal roles and in individual interactions.

4.2 Computing or IT – is that the question?

When analysing public discourses about what there is to know, learn and teach regarding digital technology or even when taking a look into educational material about the subject, a bipolarity within the broader subject can often be made out

Typical computing education courses cover the architectural and formal basics of computing. This includes programming techniques, data structures, databases and so on. It might even touch more advanced topics, such as cryptography or network protocols. The knowledge and skills developed in these courses could, for example, help describe the inner workings of instant messengers (the likes of WhatsApp, Telegram, etc.). It does not, however, cover the personal and societal functions and purposes of such digital artefacts.

In contrast, in IT or digital literacy courses, instant messaging can very likely be a proper topic. Relevant social discourses and impacts can be covered. Often, the course would include advice regarding proper and safe usage of such services, for example, covering data protection and privacy regulations, a perceived need to be accessible 24/7, maybe even identifying changes in the use of

Table 4.1 Comparing CSTA K–12 computer science standards list of concepts with the DigComp framework key components*

CSTA K–12 Computer Science Framework	Computing Systems	Networks and the Internet	Data and Analysis	Algorithms and Programming	Impacts of Computing
DigComp 2.0	Information and data literacy	Communication and collaboration	Digital content creation	Safety	Problem solving

*Light grey indicates a focus on architecture. White background indicates issues of use and/or societal impact.

language due to the characteristics of the communication media. By covering instant messaging this way, relevant aspects regarding society and – hopefully – the individual pupils are discussed. However, this typically goes without reflecting the technological aspects of the digital artefacts involved.

In existing curricula, distinctions are not as clear-cut as characterized above. Table 4.1 showcases the main contents and competences of the CSTA K–12 computer science framework (CSTA, 2017) in comparison to the DigComp 2.0 framework (European Commission, Joint Research Centre, 2016). Without going too much into detail on what the individual elements comprise, we can assign them to rather focus either on architectural aspects of computing technology (in grey) or on use and social impact (in white). It becomes clear that while a rather architecture-oriented curriculum might include some aspects of use and impact and a society- and usage-oriented curriculum may cover at least a few aspects of technical architecture, a strong emphasis on either one or the other can clearly be identified. It could be concluded that the other respective sphere of interest in both cases is covered merely as kind of an afterthought to what is considered the core of the matter.

We suggest getting rid of the strong emphases altogether, as an analysis of architectural aspects always implies the analysis of use and impact. The internal structure of digital artefacts always serves one (or more) purposes and is therefore always tightly connected with intentions. Likewise, the impact and use (as well as the usefulness) of a digital artefact cannot be thoroughly explained without understanding its inner workings. To explain one of the perspectives on technology, the other must be considered as well – at least to some degree.

4.3 The ARIadne principle as a holistic approach

In the following sections, we integrate the perspective typically subject to computing education (architecture) with the sociocultural perspective on technology (relevance). In order to integrate these perspectives, we put emphasis on the history of development which led to the current state of the artefact (genesis) and integrate the individual relations stakeholders have towards it (interactions).

Key Concepts

The ARIadne principle integrates artefact perspectives of features, architecture, relevance, genesis and interaction.

Features provide a users' manual-style description: What is an artefact capable of? Which perceivable objects of manipulation constitute this capability?

Architecture provides an objective perspective of description: What does the artefact consist of? What are its inner workings?

Relevance provides a societal perspective of description: What is the artefact used for? What does it stand for? How does it influence society? Which social discourses influence how it is used?

Genesis describes the status quo as an interrelation: Which demands by society or individuals influenced the development of the artefact? Which architectural properties did it result in? How did the existence of the artefact influence a change in societal expectations? How did these changes in turn influence the further development?

Interaction describes how one is shaped by the artefact and how one shapes the artefact: In which role is a person when using an artefact? What roles does an artefact have within the process of use?

The ARIadne principle allows integrating these perspectives into a course. The word *ARIadne* itself is also a reference to the mythical figure of Ariadne, who finds her way through a maze.

The identification of physical and sociocultural aspects of technology and their analytic discrimination is quite common in the philosophy of technology. The approach described here is explicitly based on the Duality Programme, which suggested such a distinction in the early 2000s (De Ridder, 2007; Kroes, 1998; Kroes and Meijers, 2006). With different degrees of sub-differentiations and using varying terminologies (e.g. structure and function), the Duality Programme portrays technological artefacts – of which we consider digital artefacts to be a subcategory – from two vantage points which, while being intertwined, must not be mixed up. Such a discrimination, at first glance, might appear to be a contradiction to the before-mentioned goal of integrating the purely technological and socio-psychological discourses. However, it actually should be understood as a significant step towards such an integration.

Interrelations between the two perspectives can be described and scrutinized only if those phenomena which are interrelated can be clearly identified and discriminated in the first place. Trying to cover them in their intertwined state can lead to a sublimed form of technological determinism (Bimber, 1994). Again, the example of a typical modern mobile instant messaging service can serve as an illustration. As the primary function of such a service, one might identify 'communication'. This very term, however, is used both when speaking about the service from a purely architectural point of view as well as when highlighting sociocultural aspects. However, the very same term refers to very different phenomena. In the case of architecture, when using the word 'communication', one refers to the technological property of being able to send strings of letters from one mobile device to another mobile device. When using the same word within the sphere of societal discourses, in

contrast, the meanings, purposes and intentions covered in the act of communication are in focus. When arguing within this sphere, one could, for example, discuss the impact of instant messaging services on the cultivation of friendships. Such an influence, which instant messaging services undoubtedly possess, however, clearly cannot be a property of the architecture of that messenger as nothing within the physicality of the messenger has anything to do with friendships.

While the technological and societal aspects thus have to be discriminated against each other, there is an interrelation between the two. The technological properties of the messenger services surely had their influence on what is expected in terms of human communication. If one now wanted to investigate the interrelation between architectural aspects of communication (being 'instant', providing persistent storage, revealing whether a message has been read etc.) and relevant societal aspects of communication (like to what extent one expects to stay in contact even while on vacation), this can hardly be done when mixing up the two meanings of communication. One would then not be able to distinguish what people desire when using an artefact from its physical manifestation. To avoid such a confusion, it is advisable to reflect one's usage of terminology and, if at all possible, choose different words for different concepts.

To come up with an explanation of a digital artefact which integrates the aforementioned perspectives on technology, it can prove promising not to start within one of the two perspectives themselves, but at their intersection, which in Figure 4.1 is called features.

Features describe what an artefact is used for, how it reacts to outside stimuli and which visible and manipulable properties it offers in relation to its capabilities. A major feature of an instant messenger would, for example, be its capability to send text to one or more participants of the

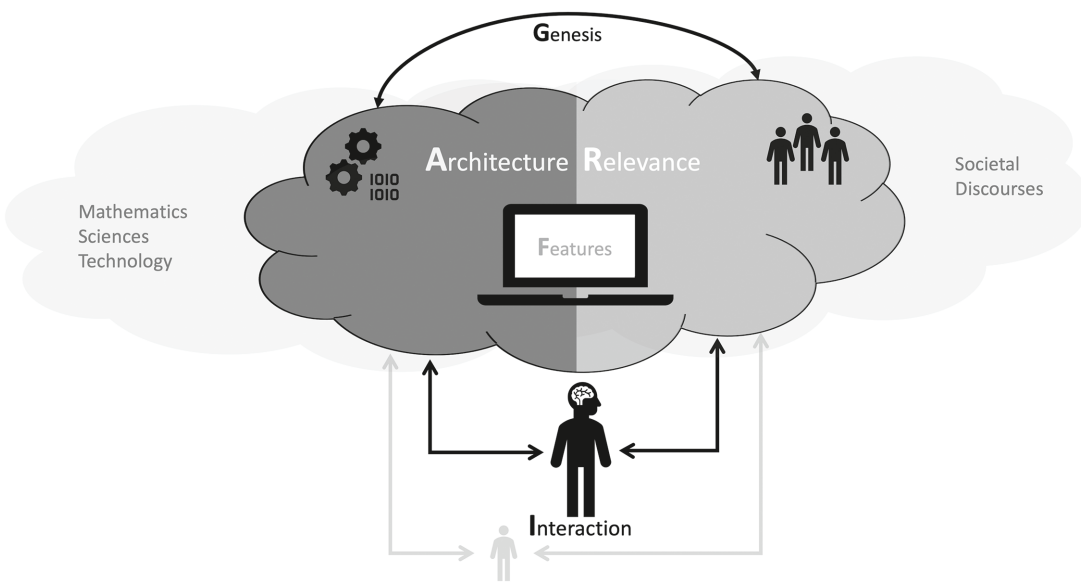

Figure 4.1 The ARIadne principle defines perspectives of description of digital artefacts; the feature perspective is extended to an architectural and a relevance perspective; these are integrated by analysing the genesis of the artefact and put into concrete context

service. For this feature, it provides, among other things, a text input field and a 'Send' button. When text is entered and 'Send' is pressed, that text is transferred to the devices of those participants of the service which have been selected before. Other features of the same artefact include the ability to check whether a message has already been seen read and the ability to have persistent access to messages of the past even when using different devices to access them.

While it is possible and sometimes even beneficial to describe an artefact solely on the feature level (e.g. in a user manual), such a perspective cannot explain how and why it even works, what it is used for, why it has been created this way and so on. To answer those questions and to understand the interrelations behind them, the world of mathematics, sciences and technology, on the one hand, and the world of societal discourses, on the other hand, must be taken into account and related to each other. However, in their entirety, both spheres are way too vast. Just to explain a digital artefact, one cannot go into the details of electrical current or explore the axioms of mathematics. Neither would it make sense to analyse the very basics of human societies and political systems. The scope of what has to be considered an integral part of the explanation has to be narrowed down. In order to do this, the features of an artefact can be made the starting point for a deconstruction (Magenheim, 2001; Magenheim and Schulte, 2006) which can uncover the necessary elements relevant to the understanding of the artefact and emphasizes their interrelations. For this deconstruction, we refer to the duality approach explained above. The essential formal-physical aspects of the artefact we call its architecture, while the relevant societal discourses we call its relevance.

When an artefact is described from the perspective of its *architecture*, its purely technological conditions and relations, its data structures and its algorithms are in focus. Descriptions within this architectural perspective are where computer scientists typically feel at home. Architectural descriptions are neutral and factual. What is described is observable as well as deterministic. When covering the same artefact from a perspective of relevant discourses, its *relevance*, other aspects are in focus, including the purpose of the artefact, the expectations which are projected upon it, aspects of its influence on society and its societal function. Within this perspective, typically interpretations instead of descriptions are being made. They are less factual in nature as they cover discourses and therefore encompass conflicting points of view. Uncovering these aspects of digital artefacts typically relies on insights and methods provided by psychologists and sociologists. However, without these aspects of relevance, the description of a digital artefact literally becomes meaningless and useless.

If one wants to figure out what an artefact represents, what it actually *is*, one has to consider the ascriptions resulting from the relevant discourses. This can be illustrated quite nicely in a gedankenexperiement in which mankind has suddenly disappeared from the face of the earth, leaving behind all of its technical artefacts. Aliens, who happened to find these artefacts, would surely consider it very difficult to figure out what the purpose of the artefacts might have been, as they can investigate only their physical properties. Without contextual information about human physiology, human needs and human cultural preferences, the actual function of these artefacts could not be determined.

On the level of mere *descriptions*, a distinction between architecture and relevance can and probably should be made. However, the two perspectives necessarily must be combined as soon as one aims to explain an artefact, as such an *explanation* is characterized by making clear why

the artefact got the way it is. Technical artefacts have not just popped up but have been developed explicitly and with a purpose over a period of time. This is true even for more complex artefacts like 'the internet', which, while not having been created by a single person or a single company at a discernible point in time, definitely has not just appeared out of thin air and surely cannot be treated as a natural phenomenon. It is clearly human-built. Therefore, the architecture of the internet exists only because people had needs and thus formulated requirements which led to its development. The actual processes behind the genesis of an artefact often are quite complex. Many stakeholders are involved. Therefore, one can conclude that no architecture would even exist without complex discourses within society. From the other angle, there is hardly any discourse within society (if at all) which is not somehow influenced by existing technology. The combination of both thoughts leads to the insight that the status quo of any digital artefact can only be described by looking into its *genesis* (its history). We conclude:

- Nothing which can be found within the architecture of digital artefacts is without history and therefore is the result of discourses of the past.
- No discourse within society is without the influence of pre-existing technology.
- Therefore, the interrelations over time need to be investigated to understand the status quo of an artefact.
- A neutral description of the architecture of an artefact essentially is a purposefully narrowed-down view disregarding that the state of the architecture is the result of past discourses.

4.4 The role of individuals: Interaction

Up to this point, the perspectives on digital artefacts were portrayed kind of actor-free. Of course, the relevant discourses do involve actual people and technology is indeed designed by human beings, yet, up to now, no individual human being was the focus of explanation. This is changed by extending our model of descriptions to include *interactions* with the digital artefact. In a school context, one of the most interesting individuals to consider will most likely be the learner (student, pupil).[1]

Individuals who use a digital artefact by interacting with it have knowledge and skills in terms of both its architecture and its relevant societal discourses. This knowledge enables them to use the artefact with some level of competency. While this relation is very individual, often certain explicit roles can be identified (see Fischer, 2002). Interactions with spreadsheet applications provide a good example for such roles. It has been found that one can identify people who see spreadsheets merely as a tool used for writing things down in an orderly manner (Borghouts et al., 2019). Others see it as an extended kind of calculator, while a third group of users create complex applications within spreadsheets. All of these roles require different sets of skills and knowledge. Following Fischer (2002), the roles and the associated self-views, world views and action schemes are not fixed but change within the process of interaction, allowing the artefacts to be shaped by individual needs and actions. At the same time, the interaction with the artefacts itself shapes one's world view, self-view and habits.

Adding the interaction aspects to the duality-based explanation of digital artefacts relate both architecture and relevance to individual perspectives on the artefact. This allows for an interaction-based duality reconstruction, where the interaction roles of pupils within different contexts are considered before developing an intervention (see, e.g., Terfloth, Budde and Schulte, 2020). Additional potentials arise when not only considering oneself or pupils as the individuals being in interaction with the device but also taking other stakeholders into account. One can then put oneself into their shoes and try to empathically understand their needs and goals by relating them to their own. This way, decisions and opinions not only are based on one's own relation towards an artefact but also include social and intersubjective deliberations.

4.5 Examples

By deconstructing a digital artefact along its architecture, its relevance, its genesis and its interaction, we have a tool which can be used for subject analysis and which can therefore be the basis for an educational reconstruction.[2] The resulting question of whether and to what extent the approach can be used to create or update a curriculum has not yet been answered. However, we can provide some practical examples, which we want to characterize briefly in order to make the orientation of the approach described here comprehensible within concrete learning processes in computer science teaching.

Parking spaces

In project courses carried out together with local schools, pupils learn that they can explore the world they live in using digital artefacts by evaluating data covering their environment. This is described in more detail in Chapter 22 about epistemic programming. In a variation of this project course, instead of having pupils gather data using sensor-equipped devices, they are provided with existing occupation data of local parking spaces, time-based data that covered several of the past years.

At the beginning of the course, pupils learn that raw data without any kind of processing does not convey any insights as correlations within the data are not yet perceivable. The data needs some kind of treatment to extract this information which is kind of hidden within it. To perform this treatment, they are introduced to data-processing techniques based on Jupyter Notebooks. Jupyter Notebooks integrate the potential of documenting the data-wrangling process with blocks of programming code which perform the actual operations. Of course, pupils in K–12 schools cannot be expected to be proficient in programming data evaluations completely by themselves. Therefore, they are provided with prefilled notebook files which already contain code examples for typical operation, such as the selection of a certain day or the accumulation of data of selected weeks or months. These code snippets can be combined and adapted for more elaborate evaluations.

Now, being equipped with a toolset (architecture) they can use to squeeze information out of the given data set, under the guidance of their teachers and tutors, pupils are asked to come up with

hypothetical relations (relevance) within the data which they can then explore using programmed data evaluations. For example, they compare occupation data during different seasons, contrast vacation weeks to typical non-vacation weeks, hypothesize on the characteristics of bank holidays and so on. As the goal of the course is to let pupils experience that they can understand the world using digital, programmable tools, the 'research questions' they are trying to answer are not predetermined by the teachers or tutors, who serve the purpose only of assistants who are there to help with formulating hypotheses and translating them into programming code.

In an extended phase of the projects, the participants were then introduced to AI methods, namely, how to train systems to make predictions based on given data sets. This triggered them to think about how to protect the environment by reducing unnecessary traffic. They came up with the idea that making drivers find a free parking space rapidly instead of having them drive around town would reduce the environmental impact. With this goal in mind, they developed a piece of software to predict future parking space occupation.

Within the course described here, programming is used to gain new insights into the world the pupils live in. They can explore it in a kind of tinkering style by combining the features of existing programming snippets. By doing so, they create new digital artefacts and adapt its architecture to their needs. By checking their hypotheses with their newly adapted artefacts, new hypotheses can and do arise. This process in itself covers the whole model of artefact explanations we described in this chapter. Students have their assumptions (relevance) and create and adapt digital artefacts (architecture) in a way it makes the necessary information perceivable (features). They do so in a cyclic fashion of getting results, updating their assumptions, adapting the artefacts, getting new results and so on. This way, they can witness the development of an artefact and its usage (genesis) in quite a short time. Ultimately, when thinking about how to convey their findings to the world and even casting them into a proper product, they consider the world views and self-views of their potential customers (interaction).

Location data in mobile communication

The initial focus of the parking spaces example described above lay in the relevance of the data and its potentials to gain knowledge about the world around us and only in a subsequent step introduced technological means and skills which are needed to actually scrutinize the data and find out what they stand for. Instead, another project about location data in a cellular network context begins by figuring out, through an analysis of the interaction of taking phone calls (features), how and why such data is needed for the cellular networks to even work (architecture). It thus starts in a context relevant and common for the learners and is then analysed in more detail. First, the students describe the features of telephone calls as part of their everyday life (relevance). Based on these descriptions, students and teachers deduce which architectural prerequisites are necessary to be able to provide these features.

The question of how the architecture works thus relates to its features which are a consequence of its relevance. Architecturally, pupils must figure out how a phone call signal is routed from sender to receiver. They learn that mobile phone networks consist of different local cells in which

a signal is delivered to base stations (visible as antennas), transmitting the signal to nearby phones. They also have to understand that to find a suitable cell, a central database (the home location register or the visitor location register) is queried to find the current location of a cell phone. By reconstructing the technical processes involved when establishing a mobile phone call, it becomes clear that without data about which phone is currently logged in to which cell, in case of a call, the whole network (meaning all cells of the network) would need to be flooded with the call information, which would not be a feasible solution in terms of technical resources. As a consequence, by analysing the infrastructure and the inner workings of a mobile phone network, the necessity to acquire location data becomes apparent.

Reflecting on this insight, potentials for the use or misuse of this kind of location- and time-based relevance are reflected upon. This reflection should also explain the existence of laws and regulations having been put into place (genesis) that limit the use of the data gathered to purposes which concern the operation of the networks themselves as well as to special cases of law enforcement. However, such rules and regulations do not exist – or are not properly enforced – in other areas where location data is gathered on mobile phones.

To give pupils a first-hand experience of the potentials of an extensive evaluation of location data (interaction), a real-life set of location data obtained from a cellular network provider is used in the course (Zeit Online, 2011). Pupils first get to see only the raw data in the form of a spreadsheet and are asked to come up with ideas on how to uncover interesting information about the person behind the data, like where he lives, where he works or what he does in his spare time. To gather this information, pupils are taught to use an interactive environment to analyse the data (architecture). They experience how simple it is to find out quite a lot about an individual as soon as a coherent body of data is created and to create a digital doppelganger. In a subsequent discussion, they reflect on what they were able to find out and, based on that, develop a sense of data awareness (relevance).

The purpose of the back and forth between architectural knowledge about the network and skills in the area of data analysis and reflections about desires, feelings and institutional regulations on the side of relevance is not to discourage pupils from using modern mobile phone technology or turning of location services at all costs, but to allow them to grasp the potentials of both use and misuse as well as be able to make informed decisions regarding their own actions in the future. It should also teach them to hypothesize on possible data involvement in modern technology more generally.

These practical examples given here show a first implementation of the approach outlined here for computer science teaching. It demonstrates that the content-oriented competencies (data storage, data processing, data analysis, algorithms etc.) do not necessarily represent new content, but rather that the content-oriented competencies become anchored within a context. Within the interaction, both the perspectives of architecture and relevance become motivated and necessary, providing the basis for the learning occasion. It can also be seen that the learning activities are always intertwined with concrete actions and reflections throughout the course, which allows the pupils to 'step out' of the process and reflect on what they were doing and then 'step in' again to implement the results of their reflections. This offers the opportunity to explicitly transfer competencies within the context and to reflect and evaluate the individual actions on a meta-level.

Key points

- When only the features and architecture of a digital artefact are considered without any integration, it is possible to describe an artefact only either technically or in terms of its use and impact without being able to explain the inherent interrelations.
- Real explanations of artefacts always require integrating the relevance perspective and the interrelation between architecture and relevance through an analysis of the artefact's genesis.
- By looking at interactions with the artefacts, the individual roles of users and the artefact in the interaction can be reflected, compared and developed.
- Combining these perspectives in the ARIadne approach allows explanations of digital artefacts and therefore their integration into computing courses.
- It also allows projects to be rooted both in architecture and relevance and, therefore, be meaningful to those carrying them out.

4.6 Summary and reflection

The approach characterized in this chapter can be used to deconstruct digital artefacts in a way that integrates different perspectives. In contrast to typical approaches towards computing education which are more canonical and typically mainly focus on the architectural aspects of technology, the process described here is explorative. In a back and forth between different perspectives of description, pupils and teachers take a journey through architecture and relevance, moderated by acts of interaction as well as by analysing the genesis of the artefacts. The examples show jumping back and forth between perspectives and connecting them is crucial to a more holistic understanding of digital artefacts. The learners recognize the interaction in terms of architecture and relevance and, based on differentiated knowledge, can uncover, analyse and evaluate interactions between architecture and relevance and hence also reflect on the intertwinedness of (individual) humans, technology and society. This way, learners get the opportunity to apply their skills and knowledge in a concrete context through the interaction with a digital artefact (Schulte and Budde, 2018). Therefore, we hope, knowledge and skills do not remain hypothetical in the sense of inert knowledge but that learners can apply what they have learned, relate to it and integrate it into their everyday lives.

Introducing such an approach in computing classes allows a reconstruction of the digital reality within the classroom and allows learning activities to be framed and guided by it. Yet, this ARIadne principle (short for architecture, relevance, interaction and a reference to the mythical figure of Ariadne who finds her way through a maze) can also be of value in a less revolutionary setting as it allows existing educational approaches to be analysed and, if necessary or appropriate, be extended. In planning or reflecting on teaching units, teachers can use the ARIadne principle to check which aspects of architecture, relevance, interaction and genesis are involved or to develop

ideas what can or should be included, probably by focussing on specific features of an artefact. Thereby, teachers can integrate perspectives and their interrelations into existing course content. In summary, we hope the principle helps to find better, that is more nuanced and balanced, answers to the questions raised at the beginning of the chapter.

For further reflection

- After reading the chapter, go back to the beginning and try to answer the questions raised there.
- The field of computing education matured to focus on stable and long-lasting concepts and competencies. See, for example, the fundamental ideas approach by A. Schwill or the concept of computational thinking described in numerous flavours. These approaches are guided by the rule to 'focus on ideas, not on artefacts!' Discuss: Doesn't the ARIadne principle increase the danger of teaching only contemporary tools and apps rather than the more persistent ideas and concepts?
- Think of a computing education lesson plan you know. Analyse and reflect on it by referring to the ARIadne principle. What, if anything at all, would you change?
- Choose a digital artefact from your environment and analyse it considering the following question: What features characterize it? How did the artefact change during its genesis? Starting from a feature, analyse its architecture, its relevance and how these are connected. What implications could this analysis have for computer science teaching?

Notes

1 Schulte brings in the perspectives of pupils by extending the idea of a didactical reconstruction towards a duality reconstruction (Schulte, 2008).
2 The concept of educational reconstruction is explained in Duit et al. (2012).

References

Bimber, B. (1994), 'Three Faces of Technological Determinism', in Merrit Roe Smith and Leo Marx (eds), Does Technology Drive History? *The Dilemma of Technological Determinism*, 79–100, Cambridge, MA: MIT Press.

Borghouts, J., Gordon, A. D., Sarkar, A., O'Hara, K. P. and Toronto, N. (2019), 'Somewhere around That Number: An Interview Study of How Spreadsheet Users Manage Uncertainty'. *arXiv preprint arXiv:1905.13072*.

CSTA (2017), CSTA K–12 Computer Science Standards, Revised 2017. https://www.doe.k12.de.us/cms/lib/DE01922744/Centricity/Domain/176/CSTA%20Computer%20Science%20Standards%20Revised%202017.pdf.

De Ridder, J. (2007), Reconstructing Design, Explaining Artifacts: Philosophical Reflections on the Design and Explanation of Technical Artifacts *(Bd. 4)*. https://books.google.de/books?hl=en&lr=lang_en|lang_de&id=ts_PtkH46ywC&oi=fnd&pg=PA9&dq=De+Ridder,+J.+Reconstructing+Design,+Explaining+Artifacts&ots=rwXS82L4kT&sig=od9dfHp8JUj4IRebveEivK6o39g.

Duit, R., Gropengießer, H., Kattmann, U., Komorek, M. and Parchmann, I. (2012), 'The Model of Educational Reconstruction—a Framework for Improving Teaching and Learning Science1', in D. Jorde and J. Dillon (eds), *Science Education Research and Practice in Europe*, 13–37, Rotterdam: Sense Publishers. doi: 10.1007/978-94-6091-900-8.

European Commission, Joint Research Centre (2016), *DigComp 2.0: The Digital Competence Framework for Citizens*, Publications Office. doi: 10.2791/11517.

Fischer, G. (2002), 'Beyond Couch Potatoes: From Consumers to Designers and Active Contributors', *First Monday*, 7 (12). https://doi.org/10.5210/fm.v7i12.1010.

Kroes, P. (1998), 'Technological Explanations: The Relation between Structure and Function of Technological Objects', *Society for Philosophy and Technology Quarterly Electronic Journal*, 3 (3): 124–34. https://doi.org/10/f2nv7z.

Kroes, P., and Meijers, A. (2006), 'The Dual Nature of Technical Artefacts', *Studies in History and Philosophy of Science Part A*, 37 (1): 1–4. https://doi.org/10.1016/j.shpsa.2005.12.001.

Magenheim, J. S. (2001), *Deconstruction of Socio-technical Information Systems with Virtual Exploration Environments as a Method of Teaching Informatics*, Chesapeake, VA: Association for the Advancement of Computing in Education.

Magenheim, J., and Schulte, C. (2006), 'Social, Ethical and Technical Issues in Informatics—An Integrated Approach', *Education and Information Technologies*, 11 (3–4): 319–39. https://doi.org/10.1007/s10639-006-9012-6.

Schulte, C. (2008), 'Duality Reconstruction—Teaching Digital Artifacts from a Socio-technical Perspective', in R. T. Mittermeir and M. M. Sysło (eds), *Informatics* Education—Supporting Computational Thinking, 110–21, Berlin: Springer.

Schulte, C., and Budde, L. (2018), 'A Framework for Computing Education: Hybrid Interaction System: The Need for a Bigger Picture in Computing Education', in *Koli Calling '18: Proceedings of the 18th Koli Calling International Conference on Computing Education Research*, 1–10, ACM.

Terfloth, L., Budde, L. and Schulte, C. (2020), 'Combining Ideas and Artifacts: An Interaction-Focused View on Computing Education Using a Cybersecurity Example'. *Koli Calling '20: Proceedings of the 20th Koli Calling International Conference on Computing Education Research*, 1–5, ACM. https://doi.org/10.1145/3428029.3428052.

Zeit Online (2011), Betrayed by Our Own Data. https://www.zeit.de/digital/datenschutz/2011-03/data-protection-malte-spitz.

5

Computational Thinking: A Competency Whose Time Has Come

Shuchi Grover and Roy Pea

Chapter outline

Chapter synopsis

Computational thinking (CT) encompasses a range of specific thinking skills for problem-solving, including abstraction, decomposition, evaluation, pattern recognition, logic and algorithm design. While what exactly is included in CT has been the topic of some debate, this chapter will consider each of the elements of CT, how the learning of these concepts and practices can be facilitated within the school curriculum and the role of CT skills in other domains.

5.1 Introduction

The twenty-first century is arguably the century of computing. Artificial intelligence has finally come of age as it becomes embedded in the transformation of work, commerce and everyday life. Big data, speech and facial recognition, robotics, internet of things, cloud computing, autonomous vehicles and 24×7 access to anyone, anywhere in the world via social media are changing how and where people work, collaborate, communicate, shop, eat, travel, get news and entertainment and, quite simply, live. Computing is also transforming industry and innovation in every discipline, becoming an integral tool that is spurring new ways of doing and thinking. In such a world saturated by computing, 'Computational Thinking' (Wing, 2006) is now recognized as a foundational competency for being an informed citizen and being successful in STEM work, one that also bears the potential as a means for creative problem-solving and innovating in all other disciplines. In this decade, systematic endeavours have gained momentum to take computer science (CS) education and CT to scale in K–12 classrooms in states across the United States and internationally. Today, CT is also seen to be moving out of academic and computing-centric realms to mentions in mainstream news stories, non-academic book titles and even as a form of (algorithmic) theatre (Grover, 2021)!

5.2 What is computational thinking?

We have witnessed over the past two decades a shift in our beliefs of what is important to learn not only in STEM subjects but also in the humanities. This shift privileges teaching higher-order critical thinking abilities fundamental in each and every domain beyond rote learning and procedural skills, in what has been designated as 'deeper learning' (Pellegrino and Hilton, 2013). US efforts around the Common Core standards for subjects such as mathematics and English language and the Next Generation Science Standards (NRC, 2013) mirror similar shifts in other countries, which emphasize disciplinary thinking and ways of knowing and being beyond rote learning. So, teaching mathematics has moved towards thinking like a mathematician; science learning now involves developing competencies for thinking like and enacting the authentic practices of a scientist.

It seems only logical, then, that educators and policymakers keen to teach computer science are attempting to privilege CT or thinking like a computer scientist, over other aspects of computing (such as learning binary arithmetic). What is 'computational thinking' anyway? How is it defined and understood?

Jeannette Wing's definition

Though somewhat opaque, Jeannette Wing's definition of CT first articulated in a 2006 Communications of the ACM article (Wing, 2006) deserves mention for capturing the collective imagination of educators and researchers worldwide and spurring global efforts to create a generation of computational thinkers. She uses 'computational thinking' as shorthand for 'thinking like a computer scientist'.

Key concept: Computational thinking

CT is the thought processes involved in formulating a problem and expressing its solution(s) in such a way that a computer – human or machine – can effectively carry it out.

Informally, CT describes the mental activity in formulating a problem to admit a computational solution. The solution can be carried out by a human or a machine. CT is not just about problem-solving but also about problem formulation (Wing, 2014).

CT is fundamentally about problem-solving using concepts and strategies most closely related to computer science. Problem formulation should be considered a key part of this problem-solving process. Since formulating the solution for the problem using CT need not involve the computer, even though the execution of the solution usually does, CT can be taught without the use of the computer. K–12 educators now aspire to teach these skills, with and without the computer, in ways that equip students to apply them in various contexts and domains and, more often than not, where a computer or computing device must carry out the solution. This is somewhat of a shift from early views of CT promulgated by Papert (1980), whose pioneering work in children and programming continues to inspire student-centred, constructionist CS curricula and pedagogies even today.

Although CT is mostly seen to be synonymous with computational problem-solving and teaching CT emphasizes skills for better preparation for a computing-centric world and economy, some framings prefer to emphasize CT learning as a means to computational creation, social engagement and participation (Kafai, 2016) or computational action (Tissenbaum, Sheldon and Abelson, 2019) that empowers the youth. Other framings emphasize an equity lens and advocate for CT learning in the context of 'critical computing' (Ko et al., 2020) through developing cultural competence (Washington, 2020), culturally relevant (Madkins et al., 2019) and community-oriented ethnocentric (Eglash et al., 2006; Lachney, 2017) pedagogies. The Raspberry Pi Foundation (2020) and Kapor Center (2021) have authored excellent guides with ideas for the classroom on culturally relevant and responsive teaching.

5.3 Elements of computational thinking (breaking it down)

What does CT mean? What thought processes does it involve? Obtaining answers to these queries will help teachers and designers to develop curricula to prepare children's CT competencies. Jeannette Wing's article and subsequent efforts to define CT – especially for K–12 education – spawned a large body of articles breaking CT down into several elements that aimed to clarify and outline what 'thinking like a computer scientist' means, including our CT 'state-of-the field' review in AERA's Educational Researcher (Grover and Pea, 2013). All these elements comprise some combination of a list of competencies most will agree are facets of the thinking processes

that computer scientists engage in when they solve problems. Keeping in mind that there is not yet an unassailable list, the elements that we find to be most comprehensive and useful to describe CT to teachers, with a few of our own tweaks, is that outlined by the British Computing at School initiative. By observing what kinds of thinking computer scientists activate when they engage in problem-solving, we find that CT encompasses the following concepts and practices. The inclusion of the 'practices' view of CT, in addition to CT concepts, is in keeping with the 'thinking like a <domain expert>' notion and describes the behaviours that domain experts engage in in the field.

Key concept: CT concepts and practices

CT concepts include the following:

1. Logic and logical thinking
2. Algorithms and algorithmic thinking
3. Patterns and pattern recognition
4. Abstraction and generalization
5. Evaluation
6. Automation

CT practices include the following:

1. Problem decomposition
2. Creating computational artefacts
3. Testing and debugging
4. Iterative refinement (incremental development)
5. Collaboration and creativity (part of broader twenty-first-century skills)

We now describe each of these along with examples set in everyday non-computing contexts as well as computing and programming contexts. Where possible, we also describe a simple example or two of how teachers might teach these concepts and practices in the classroom.

5.4 CT concepts

Logic and logical thinking

Logical thinking involves analysing situations to make a decision or reach a conclusion about a situation. Computer scientists also often use more of a formal logic framework in their work. Boolean logic is at the heart of all computing from computational circuitry to its use in software and programming to make decisions in flow of algorithmic control. As part of CT competency development, students must build analytical thinking skills by working on logical puzzles and problem-solving scenarios as well as learning formal Boolean logic through an understanding

of AND, OR, NOT (and other variants of Boolean operators) and how to construct Boolean expressions using combinations of these primitive logic elements.

Example: Boolean expressions set in real-world settings

A simple example of logical thinking might involve constructing a Boolean expression for an alarm that would ring for soccer practice on Mondays at 4:00 pm and Wednesdays at 5:00 pm, as follows:

SoccerAlarm rings IF ((WeekDay is Monday AND Time is 4pm) OR (WeekDay is Wednesday AND Time is 5pm))

Being able to reason thus with Boolean logic also translates well to game programming when games require the use of control statements involving Boolean expressions, such as, 'Game over if the player has collected all the gold coins or has no more lives left'. Of course, the program will require an additional logical check to determine whether to announce 'You won!' or 'You lost!' before it ends.

Algorithms and algorithmic thinking

Algorithms are precise step-by-step plans or procedures to meet an end goal or to solve a problem; algorithmic thinking is the skill involved in developing an algorithm. Cooking recipes are a common everyday example of algorithms (albeit less precise than what would be considered algorithms in computer science). Other common examples are route maps suggested by applications such as Google Maps or instructions for assembling a piece of furniture, instructions for knitting or crocheting a scarf and so on. In fact, the precise set of actions to get ready for school every morning could be construed as an algorithm.

Computer scientists use this concept of algorithms to devise precise solutions to problems. These solutions could be described in the form of flowcharts, pseudo-codes or a bulleted list written in an abstract everyday language that could then be coded or programmed (by the same computer scientist who creates the algorithm or by other programmers) using a programming language to be interpreted and carried out (or 'executed') by a computer. As with logic, disciplinary learning in computer science at the undergraduate level also involves a more formal study of algorithms that students may not encounter in their K–12 CS learning, involving examining aspects of efficiency, resource optimization and complexity of algorithms.

Patterns and pattern recognition

We are all familiar with the concept of patterns and pattern recognition from our early learning of shapes or mathematics topics such as multiplication and number series completion. CT includes these ideas of pattern recognition and extends the idea to problem-solving settings. Pattern

recognition in CT could lead to the definition of a generalizable solution (which also has overlaps in maths) that can leverage automation in computing for dealing with a generic situation, for example any Step n of a series no matter how large n gets. Recognizing a repeating pattern also informs how to incorporate iteration or recursion in an algorithmic solution or a functional breakdown of a problem (that also serves the cause of creating manageable and modular solutions). CT also leverages pattern recognition by examining what parts of a problem are similar to something one has already solved (or programmed) before. This is the bedrock of the powerful idea of design patterns or programming paradigms in software development.

In addition to these basic ideas of pattern recognition, computer scientists have advanced more formal use and understandings of the idea of pattern recognition in topics such as machine learning and artificial intelligence that focus on recognizing patterns in data. Pattern recognition is used in computer vision algorithms for recognizing images and faces (recall how Facebook is able to automatically 'recognize' and tag a face?) or for recommending products on Amazon, your next article on a news site or your next song using iTunes 'Genius'.

Abstraction and generalization

There is a broad consensus that abstraction is the keystone of computer science (and consequently, CT). Jeannette Wing refers to abstraction as the most important and high-level thought process in CT. It is related to several elements of CT described above. Simply put, abstraction is 'information hiding'. The act of 'black-box'-ing details allows one to focus only on the input and output. In this sense, then, abstraction provides a way of simplifying and managing complexity. It is also the ability to generalize based on similarities and differences. CT involves knowing the right types of abstractions to create and use in a computation solution.

K–12 education should strive to provide children with a sense of how computers and programming languages are also abstractions. Though the computer is a complex, physical machine made up of circuits and wires, as users of computers we interact with it through sophisticated operating systems and applications. Even computer application software developers don't need to think in terms of the physical circuitry. Programming languages used by software developers represent an abstraction of the computer that understands the constructs and keywords used in that language. These higher-level languages hide the complexity of performing operations in the more primitive instructions that are used in lower-level programming languages and ultimately the lowest level 'machine language'. A developer or an engineer at any stage typically needs to know only how to interact with one level below and what is to be seen by the next higher level. Every algorithm is also an abstraction as is every model or simulation that represents some real-world phenomenon. Every procedure defined within a program that stands for a set of instructions is also an abstraction. Data, stored in variables and data structures in programs, is the abstract 'stuff' procedures or programs act on and manipulate. They are abstract because they encapsulate and hide the details of the physical things they represent. In this important respect, computer programs are akin to more familiar algebraic equations, which also hide the details of the physical things represented in algebra equations by their variables and values.

Evaluation

Evaluation goes hand in hand with several of the elements of CT described above. Solutions to problems in the form of algorithms or abstractions in the form of programs, models or simulations must be evaluated for correctness and appropriateness based on the goal as well as constraints. While it involves analysis and analytical thinking, the idea of evaluation is grander. Solutions to problems are evaluated for accuracy and correctness with respect to the desired result or goal. There are often other grounds for evaluation. Think of the algorithm that provides directions. It could be evaluated based on any of several criteria – shortest, fastest, most scenic or other constraints such as the following: Does it take you past a grocery store or gas station where you may need to make a stop? Computer scientists dealing with complex problems and algorithms often evaluate their solutions based on efficiency constraints such as time to completion, resource usage and human factors or user experience considerations

Automation

'Computing is the automation of our abstractions' (Wing, 2008: 3718). A key part of CT, for computer science as well as computing in other domains, is working towards a solution that will be executed by a machine. Automation as a rationale to address a need that cannot be solved otherwise is often the motivation for using CT for problem-solving in the real world. In such instances, recognizing when automation is needed and what abstractions and data representations will best help develop an automated solution is a key part of CT.

At the K–12 level, even though the end goal of applying CT is not always a computational solution implemented on a machine, it is important for learners to develop an understanding of when automation is the answer to the problem – what aspects of problems are better solved by humans and which are better solved by the machines.

5.5 CT practices

The CT practices described below outline approaches that computer scientists often use when they engage in computational problem-solving.

Problem decomposition

This approach is not unique to computer scientists. It is suggested in Polya's (1957) seminal work on problem-solving in the context of mathematics. Such a method for problem-solving was enumerated as one of the rules for right thinking by Rene Descartes ([1637] 1986) in his *Discourse on Method*: 'divide each of the difficulties under examination into as many parts as possible, and as might be necessary for its adequate solution'. It is, however, a key approach in computational problem-solving. Breaking a problem down into smaller subproblems makes the problem more tractable and the problem-solving process more manageable. Examples abound in everyday life. Getting ready for

school or work usually involves getting cleaned up, getting dressed, having breakfast, packing lunch or a snack and ensuring you have the right contents in your bag as you leave. Each of these subtasks contains its own set of actions, is independent of the other and often happens in the same sequence every day. Going back to the algorithmic process of cooking and the recipe as an algorithm, one can easily see problem decomposition at play when the recipe separates the pre-preparation process of marinating or getting the ingredients together, from preparing some portion of the dish (e.g. the dressing or gravy) and the preparation of the main dish and then combining it with some post-processing steps (cooling, garnishing and such) to get the dish ready for serving.

In the context of programming, the task of breaking down a problem often leads to pieces of code being written separately. These component parts of the program need to 'come together' when the whole solution is composed. This process is simple when the different subproblems are independent of each other. Take, for example, the task of calculating the average score of an exam. The first subproblem could involve asking for user input and creating a list or array of scores; the second could address traversing the list and adding up all the scores in some aggregator variable and the final step could simply involve calculating the average by dividing the total of all scores by the number of student scores.

Creating computational artefacts

Wing's definition suggests that the goal of CT is to solve problems that can be executed by humans or computational devices. While several examples of CT described above are situated in the real world and do not involve a computer, creating solutions to be executed by a computer is often a natural end goal of CT and problem-solving. Sometimes, the computational artefact is merely a simulation or model or interactive prototype of something that will eventually be a physical artefact; at other times, the computational artefact is itself the end goal – a game or story or artefact of creativity and personal expression or software that could be used by others.

Programming is therefore seen as an especially useful platform for teaching CT since it brings together several of the elements – both concepts and practices – that are central to CT. In Grover and Pea (2013), we asserted, 'Programming is not only a fundamental skill of CS and a key tool for supporting the cognitive tasks involved in CT but a demonstration of computational competencies as well.'

Even so, it is important to observe that CT involves problem-solving and thinking competencies that can be invoked in settings outside of programming. Programming, although important, cool, interesting and fun, is but one of the possible vehicles for developing CT competencies. The current rush to focus on coding often attracts attention towards the features of the programming environment and away from the important aspects of CT that must be involved. Often these 'low-floor' programming environments allow for tinkering without the mindfulness and meta-cognition called for by deeper learning. This is akin to learning the syntax of a specific programming language (what the constructs mean and accomplish) without a deeper appreciation for the deeper CT concepts and practices that equip learners with the competencies to be used in any programming context, whatever its specific features.

Testing and debugging

Testing and debugging are integral to any kind of problem-solving (Miller, Galanter and Pribram, 1960). Evaluating one's solution for accuracy, detecting flaws in a faulty solution and fixing them are part and parcel of any problem-solving process.

Like other CT concepts and practices, testing and debugging are related to many of the other elements described here. They are part of the process of evaluating a computational solution – whether it satisfies relevant rules and assumptions, whether the solution works for boundary conditions and all relevant inputs and situations and whether it acts as expected for illogical or erroneous inputs. This also involves logical and 'if-then' analytical thinking to isolate the problem and zero in on the error. It is also integral to the incremental development and problem decomposition strategies described above.

Rigorous, systematic testing and debugging is an art and science in computing, especially in software development. Developing test cases and taking the software through its paces is a significant part of the software development process, and it is a process that itself can become automated. In the context of programming, systematic testing of the solution for correctness for the range of valid and invalid inputs is an integral competency in learning to program.

Incremental development (or iterative refinement)

This is a very common strategy used in the context of programming. Though similar to the process of problem decomposition, it focuses not so much on the idea of decomposing the problem into subproblems as it does on 'growing the solution or program' iteratively with frequent testing and debugging in between to develop improvements. This is contrasted with – and preferable to – writing large chunks of code that make it difficult to isolate the bug(s) if the solution does not work as intended. The most frequently used avatar of this approach in professional software development circles goes by the moniker 'agile development' (Martin, 2003).

Consider a simple example from robotics. Imagine a roving bot that needs to turn around when it hits an obstacle. The student could first simply code and test the movement – have the robot go forward in a straight line. Then s/he could add the obstacle collision test by making the motors stop when the touch sensor indicates a collision. Once this is tested and found to be working, the student could then add the reverse-and-turn-upon-collision code instead of simply stopping. This kind of incremental growth of the solution can be contrasted to coding all the pieces all at once and then testing. Imagine how unmanageable it can all become if the problem is more complex!

Collaboration and creativity

A couple of other elements, though not considered part of CT in earlier definitions of CT, are often described as common practices in computational problem-solving. These include collaboration and creativity. Both are acknowledged as critical competencies for a new century, but they do also have a special meaning as CT practices and in the world of computer science. Collaboration is

often fostered in K–12 computing classrooms through 'pair programming' (Williams and Kesseler, 2002) – a practice that is increasingly popular in industry. The norms of collaboration in pair programming require programmers to alternate between taking the lead on typing or reviewing code and have been shown to be beneficial to problem-solving processes. There are other forms of collaboration that are unique to CS. The division of development tasks in software engineering is necessitated by CT practices such as problem decomposition and modularization. Parallel computing has also led to the division of computing tasks and is at the heart of globally important programming paradigms such as MapReduce, which has been used for many years at Google. Other interesting forms of collaboration in the world of CS include the use of GitHub to build on one another's work in projects, crowdsourcing computer games to advance a scientific agenda (as in FoldIt and Xylem) and collaborative software development as part of the free and open-source movement that led to the creation of Linux and other systems.

Creativity as a CT practice acts on two levels – it aims to encourage out-of-the-box thinking and alternative approaches to solving problems, and it aims to encourage the creation of computational artefacts as a form of creative expression. Block-based 'open-ended' introductory programming environments such as Scratch, Alice and App Inventor have been developed with the goal of teaching creative coding and motivating learners as a conduit for teaching CT, especially in K–12 settings.

Fostering CT in the classroom and CT across subjects

Example: Logical thinking in the language arts classroom

Here is a simple problem involving logical thinking described in Grover (2009), an early ISTE article on CT ideas for teachers:

- If the Giants beat the Dodgers, then the Giants win the pennant.
- If PlayerX is out, then the Giants beat the Dodgers.
- PlayerX is out.

What is the conclusion?

Example: Logical thinking in the mathematics classroom

In a game, exactly six inverted cups stand side by side in a straight line. Each has exactly one ball hidden under it. The cups are numbered consecutively 1 to 6. Each

of the balls is painted a single solid colour. The colours of the balls are green, magenta, orange, purple, red and yellow. The balls have been hidden under the cups in a manner that conforms to the following conditions:

- The purple ball must be hidden under a lower-numbered cup than the orange ball.
- The red ball must be hidden under a cup immediately adjacent to the cup under which the magenta ball is hidden.
- The green ball must be hidden under cup 5.

Which of the following could be the colours of the balls under the cups, in order from 1 through 6?

(A) Green, yellow, magenta, red, purple, orange
(B) Magenta, green, purple, red, orange, yellow
(C) Magenta, red, purple, yellow, green, orange
(D) Orange, yellow, red, magenta, green, purple
(E) Red, purple, magenta, yellow, green, orange

Currently, computational and algorithmic thinking problems such as these are found mostly in competitions such as the Enigma Computational and Algorithmic Thinking contest run by Edfinity along with the Australian Math Trust and Bebras, but there is no doubt that students would be well-served by tackling such non-programming puzzles and problems as part of CT competency building. There are several ways of encouraging algorithmic thinking practices in learners that involve articulating precise step-by-step procedures – storyboards, an ordered set of sentences, pseudo-codes, flowcharts and the like. Even in the context of programming, expressing an algorithm in such ways before coding it into a programming language to be executed by a computer is a well-established and recommended practice. One fun exercise used in the context of robotics involves writing a set of detailed steps in plain English to verbally guide a blindfolded student partner to perform a certain task. Ideas of exception handling, iterations and conditional actions could be woven into this fun exercise.

To learners, practising these CT concepts and approaches in contexts outside programming signals the importance of the CT and problem-solving process rather than simply codifying the solution in the syntax of a programming language. Grover, Pea and Cooper (2015) describe an example of a curriculum focusing on deeper, transferable learning of algorithmic thinking skills using a pedagogy that incorporates various pedagogical ideas from the learning sciences, in addition to assessments that cover cognitive as well as affective dimensions of deeper learning.

5.6 CT within and across subjects

It is reasonable to argue that it is in all the contexts outside of CS classrooms that CT truly shines with its generativity. From music, maths, social studies, history, language arts and throughout the

sciences and engineering, curricular ideas can come alive with CT. Just as in disciplinary research in each of these fields, where CT advances both everyday practice and its innovations, there is a role for creativity in curriculum design and teaching of other subjects through the integration of CT in those classrooms, while also providing rich and varied contexts for developing CT competencies.

Bringing CT into STEM classrooms will also better prepare students for the modern landscape of the STEM disciplines. Efforts to bridge CT and STEM in K–12 science have centred mostly on building computational models and simulations to understand and study phenomena in science (e.g. Hansen et al., 2015; Hutchins et al., 2020; Sengupta et al., 2013; Wilensky, Brady and Horn, 2014) and have shown much promise. Grover, Fisler et al. (2020) and Grover, Sengupta, et al. (2020) provide excellent and varied examples of STEM and CT integration. Growing the knowledge base on how best to effect the integration of CT and STEM remains one of the imperatives for computing education research (Cooper et al., 2014; Lee et al., 2020).

The role of CT in non-STEM subjects, such as music, social sciences, visual arts, language arts and history, is manifold. Barr, Harrison and Conery (2011) and Grover (2018a, 2018b) outline examples of what this integration might look like, as does Google's Exploring Computational Thinking. Examples of documented efforts for meaningfully integrating CT and non-STEM subjects can be seen in Settle et al. (2012) and Wolz et al. (2010), among others. Fablabs, making and computational crafts also open a whole world of possibilities of CT development in the context of art and craft that involves creating tangible computational artefacts. Leah Buechley's five innovative tool designs including the Lilypad Arduino for e-textiles and 'sketch' electronics using microcontrollers and conductive ink have been found to reach audiences beyond those that robotics clubs and competitions typically attract. These tools have also supported equity-focused teaching of CS centring, ethno-computing and computational participation (Kafai et al., 2014)

Regardless of the domain, integration of CT into other subjects is often challenging for teachers and curriculum designers. Recent efforts have aimed to ease the process and lend coherence to CT integration efforts. Grover (2020) takes inspiration from Mishra and Koehler (2006)'s technological pedagogical content knowledge (TPCK) framework to integrate technology into subjects and builds on ideas from Malyn-Smith et al. (2018) to articulate 'CTIntegration', a framework that draws attention towards the intersections of CT/CS, the subject domain and pedagogical content knowledge (PCK). The framework serves as an aid for not only curriculum design and pedagogy but also teacher preparation.

Another framework called PRADA (an acronym for pattern recognition, abstraction, decomposition and algorithms) finds common ground between articulations of CT to propose a practical and understandable way of 'introducing the core ideas of CT to non-computing teachers in order to support them in infusing CT into their curricula' (Dong et al., 2019).

5.7 Summary

In a world infused with computing, CT is now being recognized as a foundational competency for being an informed citizen and being successful in all STEM work and one that also bears the potential as a means for creative problem-solving and innovating in all other disciplines. The roots

of CT in education date back to Papert's work in the 1980s that centred on children developing thinking skills through programming computers. Recent efforts on bringing CT to school education, while still inspired by that early work, have been informed by Wing's 2006 definition and call to action. Definitions and elements of CT have been broadened in the past decade to include aspects of collaboration and creativity.

CT is defined as the set of the thinking skills used by computer scientists to address a broad range of problems in computing and other domains. Learning CT, much like learning scientific and mathematical thinking, is more about developing a set of problem-solving heuristics, approaches and 'habits of mind' than simply learning how to use a programming tool to create computational artefacts. That said, programming is a key vehicle for teaching, learning, expressing and assessing CT that is unarguably also deeply engaging for students in K–12 classrooms. Much like recent movements in science and maths that have adopted a practices view to STEM learning, the key elements of CT are broken down into concepts and practices. CT concepts are commonly believed to include logic and logical thinking, algorithms and algorithmic thinking, patterns and pattern recognition, abstraction and generalization, evaluation and automation, whereas CT practices include problem decomposition, creating computational artefacts, testing and debugging and iterative refinement (or incremental development). Collaboration and creativity, now seen as cross-cutting skills for the twenty-first-century learner, are also viewed as CT practices that often acquire a unique flavour in the context of CT.

It is reasonable to argue that it is in contexts outside of CS classrooms that CT truly shines with its generativity. As Denning (2017b) points out, CT emerged from within the scientific fields – it was not imported from computer science. Computing and STEM share a deeply symbiotic relationship, and as such, mathematics, science and engineering classrooms provide perhaps the most intuitive contexts for CT learning and use. Computational modelling and simulation are concrete mechanisms for integrating computing and STEM, and can benefit the learning of both the STEM content and the development of CT skills with such an emphasis (e.g. Honey and Hilton, 2011). The role of CT in non-STEM subjects, such as music, social sciences, visual arts, language arts and history, is promising but as yet underdeveloped.

The thoughts and ideas reflected in this chapter present the current dominant framing of CT in the K–12 school education context. We adopt the disciplinary view of a thinking lens that is driving new ways of teaching and learning across all subjects. However, it is by no means the last word on this evolving topic in education and fertile field of enquiry in education research.

Key points

- Computational thinking (CT) is a key twenty-first-century skill that helps students to both understand and take advantage of computing in various domains.
- Learning CT is about learning to think like a computer scientist – developing a specific set of problem-solving skills that can be applied in any domain for creating solutions that can be executed by a 'computer' (machine or human).

- Elements of CT include concepts such as logic, algorithms, abstraction, pattern recognition, evaluation and automation. It also includes practices such as problem decomposition, creating computational artefacts (usually through programming), testing and debugging, and iterative refinement. Collaboration and creativity are broader twenty-first-century competencies that take on a special flavour in the context of CT.
- Although programming is a key vehicle to teach and learn CT, it can be taught in the classroom with or without a computer or programming.
- Bringing CT into STEM classrooms will also better prepare students for the modern landscape of the STEM disciplines; computational modelling and creating simulations are concrete mechanisms for integrating computing and STEM.
- The role of CT in non-STEM subjects, such as music, social sciences, visual arts, language arts and history, is promising but still underdeveloped.

For further reflection

- Critics of the current movement to introduce CT warn against falling into the trap of assuming that CT will help learners build thinking skills that can be transferred to other domains. Both the critics and those who make the claim against and in support of transfer ignore something we learning scientists know well. Transfer of learning across contexts does not happen automatically. Pea's own research in the 1980s showed that students who were programming in LOGO did not automatically do well in problem-solving situations in maths or in planning route scheduling. The learning sciences advocate that transfer needs to be mediated through empirically established techniques that call for, among others things, making explicit connections between the original and transfer learning contexts. For example, in past work, our classroom intervention included explicit mechanisms to mediate for and assessing transfer from block-based to text-based programming (Grover, Pea and Cooper, 2014). Have you seen any evidence in your own practice of the transfer of CT skills?
- Denning (2017a, 2017b) urges the CS/CT movement in K–12 education not to lose sight of the fact that CT emerged from within the scientific fields – it was not imported from computer science. Indeed, computer scientists were slow to join the movement. He goes on to argue that to use CT productively in science domains, one also needs the ability to design computations. Computational design is a better term to design the skill set than CT. This view closely aligns CT to the domain of computational science. Do you think CT has a role beyond computational science?
- There are some who believe that CT, if broken down into elements as described and taught through unplugged or non-programming means, will be reduced to learning thinking skills that will not necessarily translate into the abilities necessary to create computational solutions and apply CT in various domains

as per the promise. In order to learn and apply CT, students need to be working with abstractions and thinking about general solutions along with other concepts such as patterns, logical and algorithmic thinking. Writing a cooking recipe alone, although an example of algorithmic thinking, is not going to translate into providing learners the ability to develop computational solutions at the level of rigour that K–12 educators of CT and CS aim for their students. Students must experience CT in various contexts (and subjects) and through various modes and tools (unplugged, digital and programming) over the course of their elementary and secondary school journey. Where do you think that CT links to the rest of the curriculum?

References

Barr, D., Harrison, J. and Conery, L. (2011), 'Computational Thinking: A Digital Age Skill for Everyone', *Learning & Leading with Technology*, 38 (6): 20–3.

Cooper, S., Grover, S., Guzdial, M., and Simon B. (2014), 'A Future for Computing Education Research', *Communications of the ACM*, 57 (11): 34–6.

Denning, P. J. (2017a), 'Computational Thinking in Science', *American Scientist*, 105 (1): 13.

Denning, P. J. (2017b), 'Remaining Trouble Spots with Computational Thinking', *Communications of the ACM*, 60 (6): 33–9.

Descartes, R. ([1637] 1986), *A Discourse on Method: 1637*, London: Macmillan.

Dong, Y., Catete, V., Jocius, R., Lytle, N., Barnes, T., Albert, J., Joshi, D., Robinson, R. and Andrews, A. (2019, February), 'PRADA: A Practical Model for Integrating Computational Thinking in K–12 Education', in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 906–12, ACM.

Eglash, R., Bennett, A., O'Donnell, C., Jennings, S. and Cintorino, M. (2006), 'Culturally Situated Design Tools: Ethnocomputing from Field Site to Classroom', *American Anthropologist*, 108 (2): 347–62.

Grover, S. (2009), 'Computer Science: Not Just for Big Kids', *Learning and Leading with Technology*, 37 (3): 27–9.

Grover, S. (2018a, March), The 5th 'C' of 21st Century Skills? Try Computational Thinking. edSurge. https://www.edsurge.com/news/2018-02-25-the-5th-c-of-21st-century-skills-try-computational-thinking-not-coding.

Grover, S. (2018b), Helping Students See Hamlet and Harry Potter in a New Light with Computational Thinking. edSurge. https://www.edsurge.com/news/2018-11-28-helping-students-see-hamlet-and-harry-potter-in-a-new-light-with-computational-thinking.

Grover, S. (2021), 'Computational Thinking Today', in A. Yadav and U. Berthelsen (eds), *Computational Thinking in Compulsory Education: A Pedagogical Perspective*, 18–40, Abingdon: Routledge.

Grover, S., Fisler, K., Lee, I., and Yadav, A. (2020), 'Integrating Computing and Computational Thinking into K–12 STEM Learning', in *Proceedings of the 51st ACM Technical Symposium on Computing Science Education (SIGCSE'20)*, Portland, OR: ACM.

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

Grover, S., Pea, R., and Cooper, S. (2014), 'Expansive Framing and Preparation for Future Learning in Middle-School Computer Science', in *Proceedings of the 11th International Conference of the Learning Sciences*, Boulder, CO: ACM.

Grover, S., Pea, R., and Cooper, S. (2015), 'Designing for Deeper Learning in a Blended Computer Science Course for Middle School Students', *Computer Science Education*, 25 (2), 199–237.

Grover, S., Sengupta, P., Gunckel, K., Jeon, S., Dede, C., Puttick, G., Bernstein, D., Wendell, K., Danahy, E., Cassidy, M., Shaw, F., Damelin, D., Biswas, G., Dominguez, X., Vahey, P., Yadav, A., Metcalf, S., Dickes, A., Covitt, B., Farris, A. V., Berkowitz, A., Moore, J., Horn, M., Wilensky, U., Roderick, S., Stephens, L., Shin, N., Lee, I., Anderson, E., Rich, K., Schwarz, C. V. and Larimore, R. (2020), 'Integrating STEM & Computing in PK-12: Operationalizing Computational Thinking for STEM Learning & Assessment', in *Proceedings of the 14th International Conference of the Learning Sciences*, Nashville, TN: ISLS. Presentation slidedeck. https://bit.ly/STEMC-Integration.

Hansen, A. K., Iveland, A., Dwyer, H., Harlow, D. B. and Franklin, D. (2015), 'Programming Science Digital Stories: Computer Science and Engineering Design in the Science Classroom', *Science and Children*, 53 (3): 60–4.

Honey, M. A., and Hilton, M. (eds) (2011), *Learning Science through Computer Games and Simulations*, Washington, DC: National Academy Press.

Hutchins, N. M., Biswas, G., Maróti, M., Lédeczi, Á. and McElhaney, K. (2020), 'C2STEM: A System for Synergistic Learning of Physics and Computational Thinking', *Journal of Science Education and Technology*, 29 (1): 83–100.

Kafai, Y. B. (2016), 'From Computational Thinking to Computational Participation in K–12 Education', *Communications of the ACM*, 59 (8): 26–7.

Kafai, Y., Searle, K., Martinez, C. and Brayboy, B. (2014), 'Ethnocomputing with Electronic Textiles: Culturally Responsive Open Design to Broaden Participation in Computing in American Indian Youth and Communities', in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 241–6, ACM.

Kapor Center (2021), *Reimagining CS Education: A Culturally Responsive-Sustaining Framework*. https://www.kaporcenter.org/download/13902/

Ko, A. J., Oleson, A., Ryan, N., Register, Y., Xie, B., Tari, M., Davidson, M., Druga, S., and Loksa, D. (2020), 'It Is Time for More Critical CS Education', *Communications of the ACM*, 63 (11): 31–3.

Lachney, M. (2017), 'Computational Communities: African-American Cultural Capital in Computer Science Education', *Computer Science Education*, 27 (3-4): 175–96.

Lee, I., Grover, S., Martin, F., Pillai, S. and Malyn-Smith, J. (2020), 'Computational Thinking from a Disciplinary Perspective: Integrating Computational Thinking in K–12 Science, Technology, Engineering, and Mathematics Education', *Journal of Science Education and Technology*, 29 (1), 1–8.

Madkins, T. C., Martin, A., Ryoo, J., Scott, K. A., Goode, J., Scott, A. and McAlear, F. (2019, February), 'Culturally Relevant Computer Science Pedagogy: From Theory to Practice', in *Proceedings of the 2019 research on equity and sustained participation in engineering, computing, and technology (RESPECT-2019)*, 1–4, Piscataway, NJ: IEEE.

Malyn-Smith, J., Lee, I. A., Martin, F., Grover, S., Evans, M. A. and Pillai, S. (2018), 'Developing a Framework for Computational Thinking from a Disciplinary Perspective', in *Proceedings of the International Conference on Computational Thinking Education*, 5, Hong Kong: The Education University of Hong Kong.

Martin, R. C. (2003), *Agile Software Development: Principles, Patterns, and Practices*, Harlow: Pearson.
Miller, G., Galanter, E. and Pribram, K. (1960), *Plans and the Structure of Behavior*, New York: Holt, Rinehart and Winston.
Mishra, P., and Koehler, M. J. (2006), 'Technological Pedagogical Content Knowledge: A Framework for Teacher Knowledge', *Teachers College Record*, 108 (6): 1017–54.
National Research Council (2013), *Next Generation Science Standards: For States, by States*, Washington, DC: National Academy Press.
Papert, S. (1980), *Mindstorms: Children, Computers, and Powerful Ideas*, New York: Basic Books.
Pellegrino, J. W., and Hilton, M. L. (eds) (2013), *Education for Life and Work: Developing Transferable Knowledge and Skills in the 21st Century*, Washington, DC: National Academy Press.
Polya, G. (1957), *How to Solve It: A New Aspect of Mathematical Method*, 2nd edn, Garden City, NJ: Doubleday.
Raspberry Pi Foundation (2021), *Culturally* Relevant *and* Responsive Computing: *A* Guide *for* Curriculum Design *and* Teaching, Cambridge: Raspberry Pi Foundation. https://static.raspberrypi.org/files/research/Guide+to+culturally+relevant+and+responsive+computing+in+the+classroom.pdf.
Sengupta, P., Kinnebrew, J. S., Basu, S., Biswas, G. and Clark, D. (2013), 'Integrating Computational Thinking with K–12 Science Education Using Agent-Based Computation: A Theoretical Framework', *Education and Information Technologies*, 18 (2): 351–80.
Settle, A., Franke, B., Hansen, R., Spaltro, F., Jurisson, C., Rennert-May, C. and Wildeman, B. (2012, July), 'Infusing Computational Thinking into the Middle- and High-School Curriculum', in *Proceedings of the 17th ACM annual Conference on Innovation and Technology in Computer Science Education*, 22–27. New York: ACM.
Tissenbaum, M., Sheldon, J. and Abelson, H. (2019), 'From Computational Thinking to Computational Action', *Communications of the ACM*, 62 (3): 34–6.
Washington, A. N. (2020), 'When Twice as Good Isn't Enough: The Case for Cultural Competence in Computing', in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 213–19. New York: ACM.
Wilensky, U., Brady, C. E. and Horn, M. S. (2014), 'Fostering Computational Literacy in Science Classrooms', *Communications of the ACM*, 57 (8): 24–8.
Williams, L., and Kessler, R. (2002), *Pair Programming Illuminated*, Harlow: Addison-Wesley.
Wing, J. M. (2006), 'Computational Thinking', *Communications of the ACM*, 49 (3): 33–5.
Wing, J. M. (2008), 'Computational Thinking and Thinking about Computing: Philosophical Transactions of the Royal Society of London A', *Mathematical, Physical and Engineering Sciences*, 366 (1881): 3717–25.
Wing, J. (2014), Computational Thinking Benefits Society. Social Issues in Computing. http://socialissues.cs.toronto.edu/2014/01/computational-thinking/.
Wolz, U., Stone, M., Pulimood, S. M. and Pearson, K. (2010), 'Computational Thinking via Interactive Journalism in Middle School', in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 239–43. New York: ACM.

6

Learning Machine Learning in K–12

Ilkka Jormanainen, Matti Tedre, Henriikka Vartiainen, Teemu Valtonen, Tapani Toivonen and Juho Kahila

Chapter outline

Chapter synopsis

Since the 2000s, numerous practical applications of machine learning techniques have shown the potential of data-driven approaches in a large number of computing fields. The rapid diffusion of ML in apps, services and everyday gadgets has direct and significant implications on computing education at all levels. Since the mid-2010s, a quickly growing number of research and development initiatives have explored how to teach machine learning concepts also in K–12 computing education. However, as of now, the computing education research body of literature contains remarkably few studies of how people learn to train, test, improve and deploy machine learning systems. This is especially true of the K–12 curriculum space. This chapter explores challenges, opportunities and emerging trajectories in educational practice, theory and technology related to teaching machine learning in K–12 education.

6.1 Introduction: From rule-driven to data-driven computing

Data-driven society, artificial intelligence and so-called algorithms are nowadays no doubt known by most people following news, social media and public discussion. In the past ten years, the computing landscape has seen a major technological shift. Traditional programming and rule-based 'good old-fashioned artificial intelligence', which have been the driving force of automation for the past seventy years, have been joined by a variety of data-driven machine learning techniques. The much-hyped 'second machine age' (Brynjolfsson and McAfee, 2014) is based on the ability of machine learning techniques to automate many tasks that traditional, rule-based programming struggles with. In many application areas, it has turned out that for large classes of problems it is much easier to collect data sets large enough for machine learning than to figure out the rules necessary for a rule-based program (Karpathy, 2017).

Many popular examples of the latest advances in automation are based on a combination of advanced computational AI methods and traditional programming. Examples include self-driving cars, face recognition (Taigman et al., 2014), computer-based identification of tumours (Esteva et al., 2017) and the game of Go, where a computer was programmed to learn completely on its own to achieve superhuman ability in the game (Silver et al., 2017). Success and failure stories of these new technological interventions – often based on user-generated data – get publicity easily in major news outlets, and development and strategic changes of tech giants relying on AI technologies, such as Google and Facebook, are of interest for billions of users.

What is not so often talked about in public, nor taught in schools, is how everyday people leave massive amounts of data traces behind when they are using these services and how data is treated and used to build and improve the AI services. Only recently, public audiences have been made aware of phenomena such as algorithm bias.

The rapid development of AI technologies and the role of intelligent technology in society are a driving force for an evident change also in computing education. Just as previous developments in computing have triggered changes in computing education, AI technologies such as machine learning are now acting as a catalyst for change throughout the education system both in K-12 and in higher education. The focus of computing education has shifted before, new shifts will come and it has been suggested that the next frontier in computer science education research is how to teach artificial intelligence (Druga, 2018; Shapiro, Fiebrink and Norvig, 2018; Shapiro and Fiebrink, 2019).

In this chapter, we explore pedagogical, practical and technological considerations of teaching machine learning (ML) especially at the K–12 education level.

6.2 Teaching machine learning at K–12 level: Pedagogical considerations

In K–12 education, most AI-related initiatives have historically been concerned with (1) AI-based tools to support learning, (2) AI-based tools for studying learning processes and (3) AI

to support school administrative functions (Holmes, Bialik and Fadel, 2019). Adaptive learning environments, pedagogic agents, automated governance, intelligent tutoring systems and many other similar research programs have attempted to model elements of the learning situation – such as the learner, pedagogy, subject matter, context of learning and learning objects – in ways amenable to automation (Andriessen and Sandberg, 1999; Holmes, Fadel and Bialik, 2019; Williamson and Eynon, 2020). As technology and educational paradigms have changed, so have views about the merger between AI and education (Andriessen and Sandberg, 1999).

Machine learning can be considered as a subset of artificial intelligence methods, and the field can be divided roughly into two different categories. First, supervised learning engages domain experts or even end users of a system to train, let's say, an image recognition tool, with predefined samples. Training of the system is continued until the desired level of accuracy in the model is reached. Then, the model is implemented as part of a system and released to the end users. When using the second type of ML methods, unsupervised learning, an intelligent system learns important aspects of the model by itself, that is, without users' intervention. Unsupervised learning is used often, for example, in clustering problems, such as customer segmentation, and in different kinds of recommendation systems. No matter if supervised or unsupervised machine learning is used in school education, the needed pedagogy comes with new considerations when compared to traditional, rule-based programming or computational thinking education.

Traditional rule-based programming drives learners to find a solution and program an artefact which produces a correct result with all inputs. Shapiro, Fiebrink and Norvig (2018) define the traditional view of the core of computing as a collection of human-comprehensible, deterministic algorithms that can be verified. They envision two shifts away from this in the near future. They observe that machine learning models are not human-readable algorithms but opaque composites of millions of parameters. ML models are not amenable to verification efforts; however, their effectiveness can be statistically established. They further note that nearly all literature on computing education research targets rule-driven programming and, consequently, call for a major shift in the focus of computing education research to study how people learn and reason about ML systems (Shapiro, Fiebrink and Norvig, 2018; Shapiro and Fiebrink, 2019).

This paradigm shift from rule-driven programming to data-driven approach enforces teachers and learners to adopt new kinds of thinking. First, the ML process is almost always opaque at least in some parts. While the training process and the result – trained ML model – are open for inspection and learners' manipulation, the actual learning process where training data is provided to ML algorithms takes place in a black box, perhaps with some parameters available for end users to play with. Second, while the end results, such as an image recognition model, might work with most of the examples and new images presented to the system, the model might fail with obvious correct cases (false negative) or, in turn, recognize images wrong (false positive). When teaching ML in K–12, an important aspect to take into account is that the outcome is not any more deterministic, even verifiable algorithm that works correctly with all cases in its input space. Instead, students need to learn that they have succeeded with their project when the result is approximately good enough. Acceptability is no longer a binary choice; it is a degree of confidence (Tedre, Denning and Toivonen, 2021).

Key concepts

- Amount and quality of data matter the most in machine learning.
- Supervised and unsupervised learning are two main categories of machine learning.
- Supervised learning methods use training set (a representative sample of data to be learnt) to construct a model.
- Unsupervised learning methods derive meaningful patterns, such as clusters, from data without predefined samples.
- Accuracy and 'goodness' of the model are up to discretion by the end user – often it is enough that the model works 'well enough'.

6.3 What do we know this far about teaching ML in K–12?

The past few years have seen a quickly growing number of initiatives for integrating AI/ML topics in K–12 education (Tedre et al., 2021). Researchers have proposed curricular extensions (e.g. Evangelista, Blesio and Benatti, 2018; Sperling and Lickerman, 2012), frameworks for AI/ML literacy (e.g. Touretzky et al., 2019) and ethics (Ali et al., 2019; Heintz and Roos, 2021), pedagogical perspectives (Ali et al., 2019; Vartiainen et al., 2021), MOOCs (Heintz and Roos, 2021) and teacher training. Altogether, these initiatives in the K–12 context are driven by the call to provide students and teachers with the knowledge and skills needed for active agency and citizenship in a datafied world (Tedre et al., 2021).

However, building such an agenda is not straightforward, nor does it take place overnight. Traditional programming and computational thinking have been taught at schools over decades by now, but it is only during the past ten years or even less when these topics have become mainstream and a recognized part of curricula around the world. Machine learning and AI education in K-12 level is much less studied; reports from teaching experiments, tools and pedagogical solutions have started to appear only recently in the body of research literature. The researchers' and practitioners' community is still in the process of identifying, amongst other issues, (a) what to teach when talking about K-12 AI education, (b) how to teach the needed skills, (c) which tools to use to teach and (d) how to make teaching appropriate for different age groups.

6.4 Examples of potential technologies for K–12 classroom

Today, as ML-based applications have become a common part of children's everyday life, ranging from smart toys to music streaming services, attention has turned to how to teach some ML principles to children (Touretzky et al. 2019; Wong, 2020). Many modern programming languages

nowadays offer functionalities, libraries or add-on front ends to build and train ML models without exposing the underlying operations and the architecture of ML solutions to the users. Languages such as Python, R and MATLAB have been used in practical and theoretical courses on ML education in tertiary education. The trend in ML education is towards increasingly sophisticated tools for enabling practical hands-on experiences for students.

To render ML accessible and democratize the access to these technologies, the sophisticated underlying model and details of its internal implementation have been buried under layers of abstraction. Due to the complexity of many ML algorithms and the black box nature of ML's predictive models, theoretical and practical ML education use a broad spectrum of tools to soften the learning curve for state-of-the-art ML solutions. For instance, some projects have used various sets of scripting and notation languages to scaffold the black-boxed parts of the predictive models (Jormanainen and Sutinen, 2012, 2014). Other projects have used mobile robots to introduce the concepts of artificial neural networks (ANNs) and selected ML paradigms, such as reinforcement learning (Toivonen and Jormanainen, 2016; Toivonen, Jormanainen and Tukiainen, 2017). The aim of these tools is to transform a theoretical subject into a tangible, practical and explicit representation of the predictive models.

However, the tools and pedagogical approaches appropriate to each age group, learning context and educational objective are very different, and the spectrum of challenges and aims of ML education initiatives reflects these differences. ML education initiatives in K–12 are of many kinds, and often in addition to ML, concepts and skills also involve learning about fundamental principles of time and place, understanding and being able to manipulate materials and constructions (physical artefacts) as well as being able to place such artefacts into a broader historical perspective. A large number of initiatives focusing not only on ML education tools but also on pedagogical and curricular perspectives have emerged in the context of K–12 education to fill the need to teach middle-school children some basic ML concepts (Tedre et al., 2021).

These tools can provide suitable scaffolding for learners to understand the black-boxed nature of ML models. As it has been illustrated by Toivonen et al. (2020), the currently available ML tools can be used successfully in a co-design project with schoolchildren. We will present next the workflow of an image recognition project as well as the tools that can be used in it.

First step: Learning basics of image recognition with '!BB' tool

A common machine learning task is image recognition. In order to show the students that recognition does not happen by 'magic', the whole supervised machine learning process needs to be opened or glass-boxed (Toivonen, Jormanainen and Tukiainen, 2019). Essentially, image recognition of 2D images is based on extracting meaningful features from image pixel information. This information may include, for example, colour information of a pixel in RGB colour or greyscale space, pixel's relation to adjacent pixels, features of collections of pixels in the image and so on. When learning the content of an image set, machine learning algorithms read image information and extract these useful patterns from data.

Pattern recognition and image analysis are complex subjects as such, and they are not usually taught at K–12 level. To demonstrate an ML-based image recognition tool and image feature

extraction, we have built a simple web-based application (Figure 6.1) which can be used to teach basic machine learning concepts (image recognition, supervised learning, training data, feature, classifying, accuracy) to schoolchildren. We have shown (Mariescu-Istodor and Jormanainen, 2019) that it is possible to introduce and develop a machine learning method for object recognition by using knowledge that high school students attain during their normal mathematics and computing classes.

!BB ('not black-box') machine learning application (!BB, n.d.; Mariescu-Istodor and Jormanainen, 2019) uses the computer's webcam for capturing image data at live speed and classifies each frame using a simple feature-based classifier. The interface is simple, consisting of four views:

1. A small video screen of the webcam feed (Figure 6.1A);
2. The current frame as the model sees it (showing aspect ratio and fullness properties) (Figure 6.1B);
3. All currently labelled images on a 2D feature space (Figure 6.1C); and
4. A complete list of labels of currently trained classes. In addition, under view A, a small label shows the classification result for the current webcam frame and under view D, a small text box and a 'Learn' button that enables adding the current webcam image to the training set. The 2D diagram (Figure 6.1C) shows the properties of the current frame (giraffe in Figure 6.1A) with a black circle. As it can be seen in Figure 6.1C, the current giraffe frame is not in the training set (the circle is not on top of any of the trained samples), but it is close enough for other giraffe samples depicted in orange colour to be recognized as a giraffe.

An example use case of this tool could be to ask children to find sample images of the pre-trained classes and try if the tool recognizes the picture correctly. If this does not happen, it can be elaborated why the recognition failed. Another successfully used approach (Mariescu-Istodor and Jormanainen, 2019; Toivonen et al., 2022) is to ask children to draw, for example, animals that have been included in the training set. Often children's first drawings are not successfully recognized, as the artwork does not have the sufficient properties.

For instance, after finding out that a line drawing of an elephant has very low fullness (and, therefore, is easily misclassified), learners would improve their picture by colouring the elephant to improve the classification accuracy. They would make too skinny elephants wider to change the bounding box size (and to make them more elephant-like). They would draw giraffes tall and skinny-legged to affect the aspect ratio and fullness. This helps the children to understand the principles of image recognition and engages them easily in a collaborative problem-solving process when they are trying to improve the drawings.

The !BB tool presents an oversimplified model for the image classification task and uses only two properties of an image (aspect ratio and fullness). In the case of real-life image recognition applications, however, models are much more complex and feature spaces are hyperdimensional. This creates a problem in educational contexts, especially when learners do not have previous background in advanced computing or AI methods as it is almost always the case in K–12. Visualizing feature spaces beyond three dimensions can be considered, in general, a difficult task (Samek et al., 2017). Efficient image recognition tools are in many cases based on ANNs,

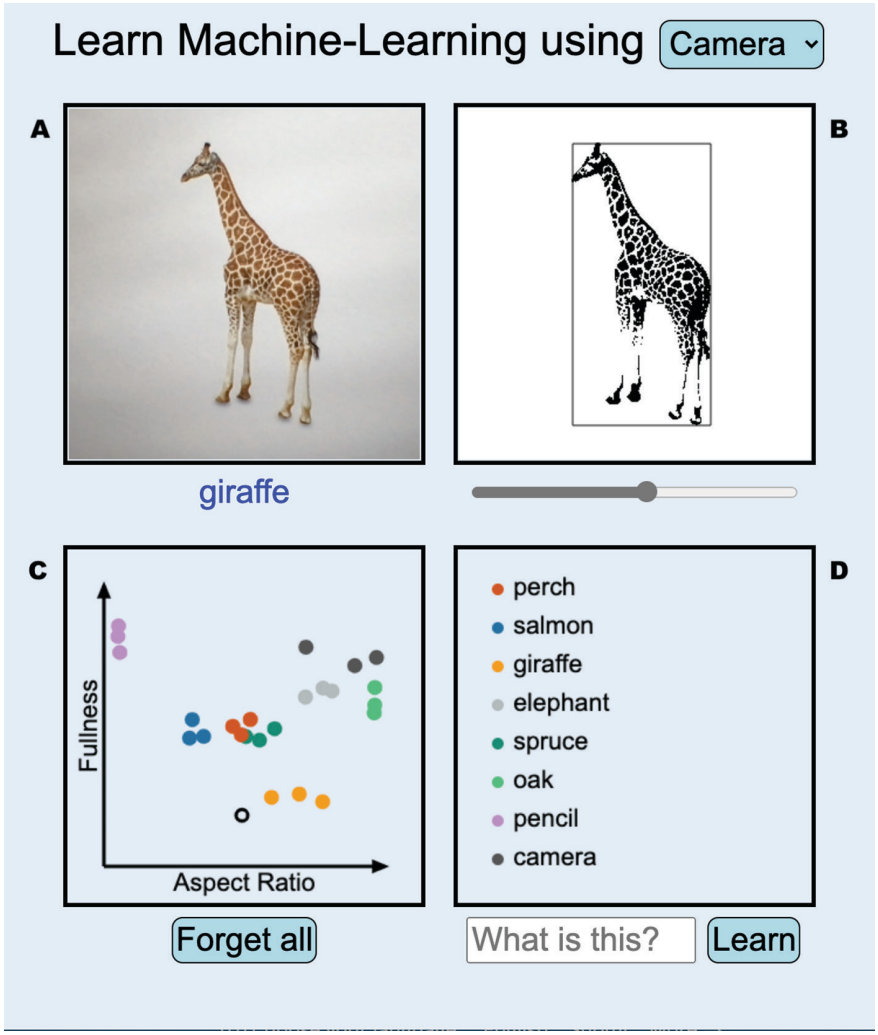

Figure 6.1 User interface of !BB image classification application

but visualizing internal structure or state of an ANN in a novice-friendly way is also difficult, and ANNs are known to be hard to trace. Nevertheless, tools such as Google Teachable Machine (GTM) provide an easy way for scaffolding after the basics of image recognition have been taught, for instance, with the !BB learning environment.

Second step: Widening the landscape in ML-based image recognition with GTM (n.d.) provides a web-based user interface for classifying images, sounds or human body poses. The system runs in a standard web browser and does not need any software or hardware add-ons, except for a web camera and microphone that are usually available in laptop and desktop computers. In the following use case description, we focus on the image recognition feature of GTM (Figure 6.2).

GTM allows users to upload image samples from their own computer or to use the computer's webcam to record training samples for different image classes. After sufficient training sets have been

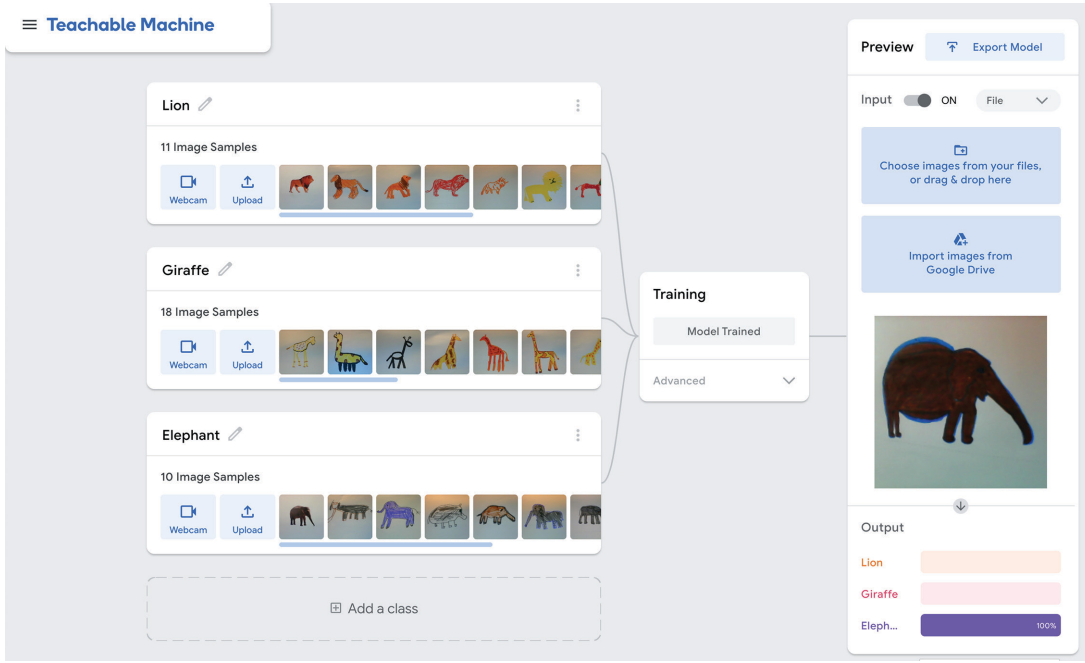

Figure 6.2 Google Teachable Machine project classifying students' animal artwork

created for the desired image classes, the users start the training process of the classifier. In contrast to !BB, the training takes place in a black box, and the users can only observe the progress of the process and see the final model. Users can adjust ML model training properties, such as epochs (how many times data samples are fed through the training model), batch size and learning rate and see key indicators of how successful the training process was, such as accuracy results and confusion matrix.

After the model is created, the students can use it with image samples that are not included in the training set to justify the quality of the end result. If the model, for some reason, fails to produce the desired classification result, the students need to improve the training sets and train the model again. This iterative process of model creation is an essential aspect when teaching how data-driven and intelligent systems work. There are several ways to leverage this process in a pedagogically meaningful way. The students can, for instance, consider if their training data has been enough or of good quality, which aspects in the training set might make the model to fail, how they could improve training data and so on. This helps them to understand what is important when working with AI and data-driven applications: It is not any more important to formulate explicit rules to control a program's behaviour. Instead, data quality, coherence and human agents' (students in the context of this chapter) decisions about 'goodness' of the model make the application work (so-called human-in-the-loop ML system; Xin et al., 2018; Toivonen, Jormanainen and Tukiainen, 2019; Toivonen and Tukiainen, 2021).

When the training process is completed and when the students are happy with the resulting classifier model (i.e. the accuracy of the model is at a desired level), the model can be exported from GTM and implemented as a part of a web or mobile application. As a promising feature of GTM, it is possible to prepare the model also for embedded systems running with low-power low-cost microcontrollers,

such as Arduino Nano 33 BLE with an embeddable camera module, such as OV7670. This opens new possibilities for integrating ML and AI education with arts, crafts and STEAM education.

6.5 Conclusions

Artificial intelligence changes our society in unforeseen ways. People are using more and more online services for work and pleasure, and a constantly growing amount of data is collected from individuals. This development also challenges the educational system to adapt to the needs of datafication and prepare students with new kinds of skills. Computing education is at the core of the paradigm shift, when rule-driven information systems are transformed to data-driven ones, where the users need to possess completely different kinds of skill sets.

School teachers are already struggling with the recent wave of integrating computational thinking into school curricula. Computing, computational thinking or programming is restricted to a rather limited number of classroom hours in national curricula or have been more or less successfully 'integrated' into other subjects. Despite the challenges, there is a clear need for understanding how ML-based and data-driven systems in people's everyday lives work.

More research and practice are needed for pedagogical models, skill progression schemes, appropriate educational technology, ethical dilemmas, domain integration and all other elements of education. Bottom-up collaboration between researchers, developers and teachers in the field is essential to understand better the bottlenecks of K–12 machine learning education and to provide relevant computing education for citizens of ever-changing datafied society.

Key points

- AI and machine learning education have an increasingly important role also in K–12 education.
- A good selection of tools and learning environments is available for machine learning education for different learning and teaching needs.
- Research on pedagogical impact, ethical aspects and curricular issues associated with K–12 machine learning education is not yet well established.

For further reflection

- Do you know what type of data, and how much, social media services collect from you when you are using them?
- What happens if you provide a classification model with more training samples? Can you reach the point where the classifier recognizes only very similar samples to the training set?

- Can you 'cheat' a trained classification model, for example a face recognition system? What features seem to be the most dominant in image recognition?
- What AI and machine learning developers should do to reduce and avoid algorithm bias?
- In your opinion, what are the most important aspects on fair and ethical use of AI services in everyday life?

References

!BB (n.d.) https://radufromfinland.com/projects/learnmachinelearning/.

Ali, S., Payne, B. H., Williams, R., Park, H. W. and Breazeal, C. (2019), 'Constructionism, Ethics, and Creativity: Developing Primary and Middle School Artificial Intelligence Education', in *Proceedings of IJCAI 2019 International Workshop on Education in Artificial Intelligence K-12 (EDUAI'19)*, 1–4, Palo Alto, CA.

Andriessen, J., and Sandberg, J. (1999), 'Where Is Education Heading and How about AI', *International Journal of Artificial Intelligence in Education*, 10 (2): 130–50.

Brynjolfsson, E., and McAfee, A. (2014), *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies*, New York: W. W. Norton.

Druga, S. (2018), 'Growing Up with AI: Cognimates: From Coding to Teaching Machines', MS Thesis, Massachusetts Institute of Technology, Cambridge, MA.

Esteva, A., Kuprel, B., Novoa, R. A., Ko, J., Swetter, S. M., Blau, H. M., and Thrun, S. (2017), 'Dermatologist-Level Classification of Skin Cancer with Deep Neural Networks', *Nature*, 542 (7639): 115–18.

Evangelista, I., Blesio, G., and Benatti, E. (2018), 'Why Are We Not Teaching Machine Learning at High School? A Proposal', *2018 World Engineering Education Forum-Global Engineering Deans Council (WEEF-GEDC)*, 1–6, Albuquerque, NM: IEEE. doi: 10.1109/WEEF-GEDC.2018.8629750.

GTM (n.d.), Google Teachable Machine. https://teachablemachine.withgoogle.com/.

Heintz, F., and Roos, T. (2021), 'Elements of AI – Teaching the Basics of AI to Everyone in Sweden', in *EDULEARN21 Proceedings of 13th International Conference on Education and New Learning Technologies*, 2568–72, IATED Academy.

Holmes, W., Bialik, M., and Fadel, C. (2019), *Artificial Intelligence in Education: Promises and Implications for Teaching and Learning*, Boston, MA: Center for Curriculum Redesign.

Jormanainen, I., and Sutinen, E. (2012), 'Using Data Mining to Support Teacher's Intervention in a Robotics Class', in *Proceedings of the IEEE 4th International Conference on Digital Game and Intelligent Toy Enhanced Learning*, 39–46, ACM.

Jormanainen, I., and Sutinen, E. (2014), 'Role Blending in a Learning Environment Supports Facilitation in a Robotics Class', *Journal of Educational Technology and Society*, 17 (1): 294–306.

Karpathy, A. (2017, November), 'Software 2.0', *Medium*. https://karpathy.medium.com/softw are-2-0-a64152b37c35.

Mariescu-Istodor, R., and Jormanainen, I. (2019), 'Machine Learning for High School Students', in *Proceedings of 19th Koli Calling International Conference on Computing Education Research*, 1–9, ACM.

Samek, W., Binder, A., Montavon, G., Lapuschkin S., and Müller, K. (2017), 'Evaluating the Visualization of What a Deep Neural Network Has Learned,' *IEEE Transactions on Neural Networks and Learning Systems*, 28 (11): 2660–73.

Shapiro, R. B., and Fiebrink, R. (2019), 'Introduction to the Special Section: Launching an Agenda for Research on Learning Machine Learning', *ACM Transactions on Computing Education*, 19 (4): 30.

Shapiro, R. B., Fiebrink, R., and Norvig, P. (2018), 'How Machine Learning Impacts the Undergraduate Computing Curriculum', *Communications of ACM*, 61 (11): 27–9.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G. V. D., Graepel, T., and Hassabis, D. (2017), 'Mastering the Game of Go without Human Knowledge', *Nature*, 550 (7676): 354–9.

Sperling, A., and Lickerman, D. (2012), 'Integrating AI and Machine Learning in Software Engineering Course for High School Students', in *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education*, 244–9, ACM.

Taigman, Y., Yang, M., Ranzato, M. and Wolf, L. (2014), 'DeepFace: Closing the Gap to Human-Level Performance in Face Verification', in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1701–8, Columbus, OH: IEEE.

Tedre, M., Denning, P. and Toivonen, T. (2021), 'CT 2.0', in *Proceedings of the 21st Koli Calling International Conference on Computing Education Research (Koli Calling '21)*, 1–8, New York: ACM.

Tedre, M., Toivonen, T., Kahila, J., Vartiainen, H., Valtonen, T., Jormanainen, I. and Pears, A. (2021), 'Teaching Machine Learning in K–12 Classroom: Pedagogical and Technological Trajectories for Artificial Intelligence Education', *IEEE Access*, 9: 110558–72.

Toivonen, T., and Jormanainen, I. (2016), 'Using JS-eden to Introduce the Concepts of Reinforcement Learning and Artificial Neural Networks', in *Proceedings of 16th Koli Calling International Conference on Computing Education Research*, 165–9, ACM.

Toivonen, T., Jormanainen, I., Kahila, J., Tedre, M., Valtonen, T., and Vartiainen, H. (2020), 'Co-Designing Machine Learning Apps in K–12 with Primary School Children', *2020 IEEE 20th International Conference on Advanced Learning Technologies (ICALT)*, 308–10.

Toivonen, T., Jormanainen, I., Tedre, M., Mariescu-Istodor, R., Valtonen, T., Vartiainen, H., and Kahila. J. (2022), 'Interacting by Drawing: Introducing Machine Learning Ideas to Children at a K–9 Science Fair', *CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI '22 Extended Abstracts)*, 1–5, New York: ACM.

Toivonen, T., Jormanainen, I., and Tukiainen, M. (2017), 'An Open Robotics Environment Motivates Students to Learn the Key Concepts of Artificial Neural Networks and Reinforcement Learning', *International Conference on Robotics and Education (RiE)*, 317–28.

Toivonen, T., Jormanainen, I., and Tukiainen, M. (2019), 'Augmented Intelligence in Educational Data Mining', *Smart Learning Environments*, 6 (10): 1–25.

Toivonen, T., and Tukiainen, M. (2021), 'The Power of Human–Algorithm Collaboration in Solving Combinatorial Optimization Problems', *Algorithms*, 14 (9): 253.

Touretzky, D., Gardner-McCune, C., Martin F., and Seehorn, D. (2019), 'Envisioning AI for K–12: What Should Every Child Know about AI?' in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*, vol. 1, 9795–9.

Vartiainen, H., Toivonen, T., Jormanainen, I., Kahila, J., Tedre, M. and Valtonen, T. (2021), 'Machine Learning for Middle Schoolers: Learning through Data-Driven Design', *International Journal of Child-Computer Interaction*, 29: 100281.

Williamson, B., and Eynon, R. (2020), 'Historical Threads, Missing Links, and Future Directions in AI in Education', *Learning, Media and Technology*, 45 (3): 223–35.

Wong, K. C. (2020), 'Computational Thinking and Artificial Intelligence Education: A Balanced Approach Using Both Classical AI and Modern AI', in *Proceedings of International Conference on Computational Thinking Education*, 110–11.

Xin, D., Ma, L., Liu, J., Macke, S., Song, S. and Parameswaran, A. (2018), 'Accelerating Human-in-the-Loop Machine Learning: Challenges and Opportunities', in *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning (DEEM'18)*, 1–4, New York: ACM.

Part 2

Computing for All: Equity and Inclusion

7

Introduction to Part 2

Nicol R. Howard

Computer science (CS) impacts just about every facet of society, and its influence on varying fields has increased efforts to make CS educational opportunities accessible to all learners. Movements such as Computer Science for All (CS for ALL) have helped to expand course offerings in upper grades and influence increased levels of CS integration in classrooms with young learners. Efforts to engage all learners in CS have evolved for various reasons from the hope to broaden participation to increasing job opportunities and high wages associated with the field. Yet inequities persist in CS education policies, practices and classroom environments.

Greater awareness of the inequities in CS education has resulted in research and practice focused on justice-oriented approaches to preparing learners and transforming learning environments. More specifically, researchers and practitioners have increased their efforts to better understand how to strengthen CS experiences for minoritized learners (i.e. racialized, gendered, multilingual, identifying as having a dis/ability) to ensure sustainable and inclusive CS learning environments that consider the intersectional identities and needs of all learners. In different areas of the world, the terms used to describe these various efforts and the communities they serve to benefit may differ (e.g. minoritized in the United States, diversity in the UK); however, the collective goal is to ensure equity. The authors of the chapters in this section invite readers to reflect on the issues of diversity, equity, inclusion and justice in their own geographical contexts and how these may influence broader sociopolitical considerations.

The chapters in this section were written by educators and researchers who focus their work on justice-oriented and equity-centred CS education for all. Their work calls for reducing inequities and transforming CS learning and environments into more inclusive experiences that integrate knowledges that centre students' identities and communities. Both technological, as well as pedagogical issues, and manifold implications for making CS opportunities accessible for all learners will be presented by the authors of the chapters in this section.

The section begins with 'Equity and Inclusion in Computer Science Education: Research on Challenges and Opportunities' by Jill Denner and Shannon Campe. Their chapter begins by outlining existing research on equity and inclusion in computer science education. We learn about the various factors that limit representation in CS education and why it is important to purposefully

address equity and inclusion. Denner and Campe present opportunities for increasing diversity and offer five different types of strategies that teachers can use to support equity and inclusion in CS classrooms.

Tia C. Madkins and I examined the ways in which educators can facilitate justice-oriented learning practices in 'Engaging Culturally Relevant and Responsive Pedagogies in Computer Science Classrooms'. In this chapter, we provide a general understanding of the current state of the field related to CS education and advocate for using equity pedagogies (e.g. culturally relevant pedagogy, culturally relevant computing) through a justice-oriented lens. We offer practical examples and guidance on how to use equity-centred pedagogies. Our belief that educators and researchers need to understand what it means to use a justice-oriented equity lens in CS teaching and learning informs the recommendations we offer on how to engage equity pedagogies in CS classrooms.

In 'Increasing Access, Participation and Inclusion within K–12 CS Education through Universal Design for Learning and High Leverage Practices', Maya Israel, Latoya Chandler, Alexis Cobo and Lauren Weisberg aim to answer 'what do we mean by Computer Science for All?' and 'how can we plan CS in a manner that proactively includes all learners?' We learn how transforming inclusive CS education relies on supporting teachers with their understanding of the intersectional identities of students. The authors offer strategies for including all learners in CS education through the use of inclusive practices, such as Universal Design for Learning (UDL) and High Leverage Practices (HLPs), as well as on the intersection of UDL with culturally responsive and sustaining pedagogies.

8

Equity and Inclusion in Computer Science Education: Research on Challenges and Opportunities

Jill Denner and Shannon Campe

Chapter outline

Chapter synopsis

This chapter introduces research on equity and inclusion in computer science education. Topics include why it is important to intentionally address equity and inclusion in computer science education, the factors that limit representation, opportunities for increasing diversity according to theoretical perspectives and empirical research and five different types of strategies that teachers can use to support equity and inclusion in computer science classes.

8.1 Introduction: Setting the stage

Enthusiasm for computer science (CS) education in K–12 is growing worldwide, but not all students are participating equally. Groups that are under-represented include females, students with learning differences/disabilities and, depending on the country, students from certain racial or ethnic minority groups. This chapter describes strategies that teachers can use to increase equity and inclusion and help to address these gaps.

Globally, when conversations turn to equity and inclusion in CS education, the focus is usually on gender. This is because in most countries men outnumber women in computing fields. There are only six countries where women make up more than 30 per cent of tertiary graduates in information and communication technologies: Canada, Estonia, India, Indonesia, Turkey and Mexico (Huyer 2019). Among the bachelor's degrees awarded in computer and information science in 2017–18 in the United States, only 20% were awarded to women (National Center for Education Statistics, 2019) and only 18 per cent of students enrolled in CS in the UK in 2018/19 were female (HESA, 2020). However, there is gender parity in some countries, such as in enrolment in CS studies in Saudi Arabia and in the IT workforce in Malaysia (Frieze and Quesenberry, 2019).

Data on the participation of students across racial and ethnic groups is available primarily in the United States and shows persistent gaps. In CS undergraduate programs in the United States, Black and Hispanic/Latinx students are under-represented in enrolment and have lower rates of retention and degree completion compared to white and Asian students (Duran et al., 2021; Zweben, 2019). And while recent years have shown an increase in the number of Native American/Alaskan, Black, Hispanic/Latinx and Native Hawaiian/Pacific Island students taking AP CS Principles tests, there has been a decline in participation among the latter three groups in taking AP CS A tests (Code.org, 2021), and pass rates continue to be much lower among Black and Hispanic students than among white and Asian students (Ericson 2021).

These findings do not capture the variation within racial categories, so an intersectional lens is needed to understand inequities in computing education (Warner et al., 2021). While fewer than half of Black and Hispanic students (46 per cent each) and 52 per cent of white students say their schools offer dedicated CS classes, access is the lowest among rural students and those from low-income households (Gallup, 2021). Rather than focusing on demographic categories, it may be more useful to consider how social identities expose people to overlapping and interdependent systems of discrimination or disadvantage (Charleston et al., 2014; Kvasny, Trauth and Morgan, 2009). Included in this lens needs to be whether a student has a disability or learning difference.

In the United States, 15 per cent of K–12 students have a disability; the most common are in areas of learning, speech and language, health impairments, developmental delay and emotional disturbance (Ladner and Israel, 2016). Although grouped under the umbrella of disabilities and learning differences, many of these students do not have an intellectual disability and bring an ability to approach and solve complex problems in innovative ways that may be particularly beneficial for learning CS (Wille, Century and Pike, 2017). However, there has been limited

attention to the creation of inclusive learning environments for students with disabilities (Stefik et al., 2019).

To address these gaps, there is a growing movement to integrate CS into schools, where it has the potential to reach all students. In countries like Russia, there has been a system for teaching CS in schools since 1985 (Khenner and Semakin, 2014). And in the UK, government officials have begun to publicly support CS education efforts (Brown et al., 2014). However, these efforts have met resistance. For example, a nationwide effort to integrate CS into the schools in New Zealand required public media campaigns to address concerns about what it was replacing (Bell, Andreae and Robins, 2014). In India, while the majority of students feel that CS should be a compulsory subject, over half of the teachers do not agree (Raman et al., 2015). And in Switzerland, most pre-service teachers agreed that a new mandatory CS education class was important, but the women were less interested in learning CS and more likely to view it as hard or boring (Repenning et al., 2019). While most parents in the United States want their children's schools to offer CS (Google Inc. and Gallup Inc., 2015) and most Americans believe that it is as important to learn CS as it is to learn reading, writing and mathematics (Horizon Media, 2015), integration across states and school districts varies greatly.

Despite the growing movement to offer CS to K–12 and Key Stage 1–5 students worldwide, the focus is squarely on access, not on equity. But K–12 and Key Stage 1–5 CS education will only lead to more diverse classes, majors and ultimately the workforce when it is equitable and inclusive. The Capacity for, Access to, Participation in, and Experience (CAPE) framework developed by Fletcher and Warner (2021) shows how equity can become an intentional part of the educational ecosystem starting with building the capacity of districts and teachers to offer inclusive instruction. Thus, CS teachers must understand the content as well as diversity issues and pedagogical strategies for inclusive CS education (Delyser et al., 2018). However, less than 60 per cent of CS teachers in the United States feel prepared to address equity in the classroom; that number is lower among white and elementary school teachers (Koshy, et al., 2021). This chapter provides a resource for teachers to increase diversity and equity at all grade levels in CS education by creating inclusive learning experiences and institutions.

Key concepts

Diversity refers to the representation of different kinds of individuals in computer science, across languages, learning styles and other differences as well as a range of social or cultural groups (across race, ethnicity, class, gender, sexual orientation, dis/ability status, etc.).

Inclusion involves an active and intentional engagement with diversity such that a range of individuals are able to fully participate in computer science education.

Equity requires the creation of opportunities for historically under-represented populations to have equal access and participate in computer science education; it requires an asset-based approach and an understanding of why disparities exist.

8.2 Why are certain groups under-represented in computing?

There are a range of factors that perpetuate inequities in computing. Most efforts focus on individual-level factors, but there is increasing attention to the role of relational and institutional factors (Denner, Martinez and Thiry, 2016). Key research and theory on the role these factors play in equity in CS education are summarized below.

Individual factors

Expectancy-value theory describes the role of students' belief systems in their educational choices, including their expectations for success and the extent to which they value a field like computing (Eccles, Wigfield and Schiefele 1998). For example, a student's confidence and perceptions of their mathematics and problem-solving abilities affect their decisions about whether or not to pursue computing (Hong et al., 2015). Among girls, decisions to pursue computer science are influenced by their interest in mathematics, problem solving, creativity and design as well as their curiosity about how technology works (Cooper and Heaverlo, 2013; Denner, 2011). Girls' interest is also influenced by the extent to which they think positively about CS and related careers and see their potential for social impact (Hong et al., 2015). While interest and confidence can initially motivate students to seek out opportunities to learn computing, relational support and institutional opportunities are needed to sustain that interest.

Relational factors

Expectancy-value theory also suggests that achievement goals are influenced by culturally based beliefs and by expectations of teachers and other influential adults (Eccles, 2007). For example, in Malaysia the lack of a gender gap is likely due to the cultural norm that computing is not a masculine field (Othman and Latih, 2006) and to girls being encouraged and given access at an early age (Sien et al., 2014). In the United States, disinterest in computing fields is partially a result of negative stereotypes that students from some groups are less interested in CS (Master, Meltzoff and Cheryan, 2021) or less likely to be successful – a belief held by parents, teachers and administrators (Google Inc. and Gallup Inc., 2015; Margolis et al., 2008). These beliefs result in less encouragement of female students to pursue STEM or CS (Google and Gallup, 2016; Muenks et al., 2020). Similarly, misperceptions about the intellectual capacity of students with disabilities can limit what educators expect of them (Ladner and Israel, 2016). These stereotype-based messages sent by adults are often small, subtle and cumulative in nature and result in lower self-concept and expectations for success in computing fields (Else-Quest, Mineo and Higgins, 2013). While high expectations and encouragement can increase students' interest in CS, they also need access to inclusive CS learning environments.

Institutional factors

Institutional structures at multiple levels determine who has access to CS and what their experience is in the classroom. Country- and state-wide initiatives have resulted in increased access in countries like Israel, where extensive CS education in high school created more equal numbers of women and men taking the CS exam (Armoni and Gal-Ezer, 2014). But resources vary across schools and districts; in the United States, rural schools and schools that serve large numbers of low-income and Black, Latinx, Indigenous and Pacific Islander students have the fewest CS opportunities (Koshy et Al., 2021; Margolis et al., 2008). And students with disabilities cannot be fully included when there are no accessible tools, such as screen readers for visually impaired students (Ladner and Israel, 2016). How a classroom is decorated influences all students' sense of belonging (Master, Cheryan and Meltzoff, 2016). There are growing efforts to institutionalize the use of racially just and inclusive strategies through ongoing training and support for teachers (Goode et al., 2021).

In summary, whether or not a person chooses to pursue and persist in a computing field is influenced by a multitude of factors that include personal beliefs, messages from other people and the environment and the extent to which the learning environment provides the necessary access, tools and scaffolds that a student needs to succeed both in and beyond the classroom.

8.3 Opportunities for change

In this section we describe a framework for the types of change mechanisms that can be used to address individual, relational and institutional factors that perpetuate a lack of diversity in CS education. It builds on Liben and Coyle's (2014) taxonomy of intervention goals to address the gender gap in STEM that includes five change mechanisms based on theories and research on gender development. Table 8.1 outlines these change mechanisms by adapting the terms used by Liben and Coyle. Each one is described, along with intervention strategy examples, in the following sections.

Table 8.1 Five intervention change mechanisms for increasing representation in computer science

Change Mechanisms	Definition
Remediate	Fix individual qualities that are considered important for success in CS (e.g. increase confidence, boost skills)
Revise	Modify pedagogy, tools and the environment so they better fit a diverse group of students
Refocus	Highlight how a range of interests can be compatible with success in CS
Recategorize	Shift thinking about certain identities being incompatible with CS
Resist	Work to challenge stereotypes, biases and discriminatory practices

Remediate

Strategies to promote this change mechanism focus on students' skills and/or their beliefs about their abilities. In Russia, a nationwide curriculum was developed to increase computing knowledge (e.g. algorithmic thinking) starting in elementary school (Khenner and Semakin, 2014). And specific strategies, like teaching children to program computer games, have been used to increase students' understanding of programming concepts and their problem-solving capacities with results replicated in countries as diverse as Turkey, Taiwan and the United States (Akcaoglu, 2013; Al-Bow et al., 2009; Wang and Chen, 2011). US-based studies find it is possible to increase students' confidence, interest and skills in CS through positive reinforcement and encouragement from family, peers and other adults (Hong et al., 2015) as well as pair programming, where two students work side by side on one computer (Denner et al., 2014). A study of Scottish college students in an introductory programming class found that helping students develop performance attributions that focus on factors that they can change increased their growth mindset and test scores (Cutts et al., 2010). While efforts to 'remediate' individuals are incomplete without regard to the relational and institutional contexts of learning, changes in students' beliefs and skills are commonly used to measure success.

Revise

These approaches focus on developing inclusive pedagogies and tools to make CS a better fit for a range of students. Effective CS pedagogy includes hands-on project-based learning that leverages prior experience, uses peer collaboration and shows how CS is relevant to students' lives (Happe et al. 2021; Shaw, Fields, and Kafai 2020). Strategies for increasing the relevance include showing students how computer science can be used to address needs in their schools and communities (Denner et al. 2015; Cheryan et al. 2009). With the right supports and scaffolds in place, pair programming (see example activity) can be an effective collaborative strategy for supporting students with a range of confidence and experience in CS classrooms (Campe, Green, and Denner 2019; Lewis 2011). The Universal Design for Learning (UDL) approach aims to accommodate a range of learning differences and includes inclusive pedagogical strategies, such as using a screen reader and offering activity-oriented tasks that allow the use of multiple senses to learn (Hansen et al. 2016; Ladner and Israel 2016; also see Chapter 10 by Israel et al.). Recent efforts have made popular US curriculum like CS Principles accessible for blind and visually impaired students and include training teachers in accessibility (Stefik et al., 2019).

Example: Using pair programming to promote equity and inclusion

Working with a partner while learning to program a computer can increase students' engagement and learning when it is managed effectively. Start by asking students to choose two to three people they are willing to work with and use that information to

match them to a partner with similar prior experience and confidence. Build the pair's communication and rapport through non-computer-based activities such as 'Draw what I say'. Demonstrate what both effective and ineffective communication looks like through videos and role plays. Reinforce effective communication with public acknowledgments, such as 'Pair of the Week'(Campe, Green and Denner 2019).

Refocus

This set of strategies aims to challenge stereotypes about the kind of skills or interests that are compatible with a career in computing. While views are changing, many young people still believe that computer scientists are men who work alone all day on a computer (Hansen et al., 2017). But showcasing creativity and design can be a good strategy for engaging students in computing. Programs like Digital Youth Divas teach girls to design everyday artefacts (jewellery, hair accessories, music) and include collaboration, critique, circuitry, coding and fabrication (Pinkard, Martin and Erete, 2020). Digital storytelling is often used as a mechanism for the designer to interpret and share the world around them and to highlight voices that may not be heard otherwise (Peppler et al., 2014; see Chapter 4 by Schulte et al. for examples of the interrelation between digital technology and everyday life and society). Workshops such as Skins incorporate digital design, animation and game programming that students use to modify digital games based on the traditional stories of their Native American community (LaPensée and Lewis, 2011). E-textiles combine computation and crafting with the use of fabric, conductive thread, circuits and lights to create artefacts for practical or artistic use and have been shown to increase interest in computing, especially for girls (Jayathirtha and Kafai, 2019). Media computation builds on students' interest in sound, photography and video while introducing them to computing concepts and skills (Guzdial, 2013). However, while media computation succeeds in changing how much students value computing and retains more students than traditional computing classes, it does not necessarily increase long-term participation in computing because it does not directly address relational or systematic factors (Guzdial, 2013).

Recategorize

Most strategies in this category aim to challenge stereotypes about who is good at (and who should pursue) computer science. For example, Craig (2014) describes an intervention in Australia, where the media and local speakers challenge stereotypes that CS is not for girls, raise awareness among teachers about CS careers and offer girls opportunities to shadow CS college students and attend a Girls in Computing day. In the United States, programs that connect Black and Latinx girls with same-sex role models and near-peer mentors challenge negative stereotypes about computer science careers and increase interest and intention to pursue these types of careers (Denner, Martinez and Thiry, 2016; Erete et al., 2021; Dasgupta, Scircle and Hunsinger, 2015).

A smaller number of programs send the message that computing is compatible with a range of cultures and identities. The TECHNOLOchicas campaign highlights the interests and needs of young Latinas through stories of Latinas studying or working in technology (Fernandez and Wilder, 2020). Additional strategies include integrating knowledge that is relevant to students' identities and communities within activities that are designed to promote computational learning. These culturally responsive approaches go beyond tech literacy to centre culture, community and intersectional identities to explicitly challenge stereotypes about who belongs in computing (Scott, Sheridan and Clark, 2015) and also show the connection to more current cultural references (Eglash et al., 2013).

Resist

A justice-centred approach to computing advocates for addressing the systemic factors that perpetuate inequity in CS education (Lachney, Ryoo and Santo, 2021). This includes developing students' critical consciousness about how institutionalized systems of power, bias and oppression perpetuate inequity in CS. Examples include fostering conversations about social justice, race and gender and creating opportunities for students to use CS to build solutions that address inequities in their school and community (Denner et al., 2015; Jenson, Dahya and Fisher, 2014; Scott and White, 2013; Vakil, 2014). Going beyond the classroom, Erete et al. (2021) use a transformative approach that involves knowing the institutional histories so as to not replicate injustice and involving families to challenge local policies that limit access. Madkins, Howard and Freed (2020) describe how teachers can use equity pedagogies and present a new framework that incorporates these elements and describes the steps that teachers can take to implement a culturally responsive-sustaining pedagogy in CS (Kapor Center, 2021; also see Chapter 9 by Madkins and Howard).

8.4 Conclusion

While an increasing number of studies have aimed to understand when and why students choose not to study CS, the focus has been largely on the gender gap. A recent global snapshot of CS education shows that one challenge to understanding equity and inclusion is the wide variation in the terms used and the learning outcomes targeted across countries (Hubwieser, Armoni and Giannakos, 2015). However, there remains a need for studies of racial/ethnic minority groups in countries besides the United States as well as studies that account for students' multiple intersecting identities.

Liben and Coyle's (2014) intervention types provide a useful framework for K–12 educators to be intentional in how they support equity and inclusion as they teach computer science. While it may not be possible to address all five mechanisms at once, it is important to recognize which are being prioritized and why for a given group of students. By organizing efforts using the framework, it will become more clear which strategies are not yet being addressed. Ultimately, it is important

to develop and test strategies for all five change mechanisms for the needed progress on increasing diversity in computer science to take place.

8.5 Recommendations

Most large-scale efforts begin by working to incorporate computer science into schools with a focus on building individual skills and competencies. In this chapter we advocate for incorporating relational and institutional goals early in the process. Teachers can use the framework to evaluate whether their strategies are inclusive, starting with looking at their own students and institutional context and assessing which parts of the framework are in need of more immediate attention and which parts they have already succeeded in addressing. Teachers can use the framework to advocate for resources, such as training in inclusive pedagogies and access to tools that are inclusive of different types of learners. They can also use the framework to identify gaps that can be filled by forming partnerships with colleges, industry or out-of-school programs. These actions can lead to more equitable opportunities and more inclusive learning environments.

Key points

- A focus on equity and inclusion is an important part of quality computer science education.
- The challenges to fostering an equitable and inclusive learning environment exist at the institutional, relational and individual levels.
- Five different types of change mechanisms can be used to address the lack of diversity in computer science: remediate, revise, refocus, recategorize and resist.
- The most common strategies focus on remediating individuals or revising the pedagogy or tools, and the least common strategy is helping students develop a critical consciousness in order to transform the structural issues that perpetuate the disparities.
- Efforts to incorporate CS into schools must go beyond individual goals to also address relational and institutional goals.

For further reflection

- For each of the five change mechanisms described in this chapter, list two to three activities that you could do with students.
- Considering the factors that prevent equity and inclusion in computer science education, reflect on which challenges would be the most amenable to change in your own classroom.

Acknowledgements

This work was supported by Grant 1923606 from the National Science Foundation. The authors are grateful to Heather Bell for her help with references and formatting.

References

Akcaoglu, M. (2013), 'Cognitive and Motivational Impacts of Learning Game Design on Middle School Children', PhD dissertation, Michigan State University.

Al-Bow, M., Austin, D., Edgington, J., Fajardo, R., Fishburn, J., Lara, C., Leutenegger, S. and Meyer S. (2009), 'Using Game Creation for Teaching Computer Programming to High School Students and Teachers', *ACM SIGCSE Bulletin*, 41 (3): 104–8. https://doi.org/10.1145/1595496.1562913.

Armoni, M., and Gal-Ezer, J. (2014), 'Early Computing Education: Why? What? When? Who?' *ACM Inroads*, 5 (4): 54–9. https://doi.org/10.1145/2684721.2684734.

Bell, T., Andreae, P. and Robins, A. (2014), 'A Case Study of the Introduction of Computer Science in NZ Schools', *ACM Transactions on Computing Education*, 14 (2): 1–31. https://doi.org/10.1145/2602485.

Brown, N., Sentance, S., Crick, T. and Humphreys, S. (2014), 'Restart: The Resurgence of Computer Science in UK Schools', *ACM Transactions on Computing Education*, 14 (2): 1–22. https://doi.org/10.1145/2602484.

Campe, S., Green, E., and Denner, J. (2019), *K–12 Pair Programming Toolkit*, Scotts Valley, CA: ETR.

Charleston, L.J., Adserias, R. P., Lang, N. M., and Jackson, J. (2014), 'Intersectionality and STEM: The Role of Race and Gender in the Academic Pursuits of African American Women in STEM', *Journal of Management Policy and Practice*, 2 (3): 17–37.

Cheryan, S., Plaut, V. C., Davies, P. G., and Steele, C. M. (2009), 'Ambient Belonging: How Stereotypical Cues Impact Gender Participation in Computer Science', *Journal of Personality and Social Psychology*, 97 (6): 1045–60. https://doi.org/10.1037/a0016239.

Code.Org (2021), Dig Deeper into AP Computer Science. https://code.org/promote/ap.

Cooper, R., and Heaverlo, C. (2013), 'Problem Solving and Creativity and Design: What Influence Do They Have on Girls' Interest in STEM Subject Areas?' *American Journal of Engineering Education (AJEE)*, 4 (1): 27–38. https://doi.org/10.19030/ajee.v4i1.7856.

Craig, A. (2014), 'Australian Interventions for Women in Computing: Are We Evaluating', *Australasian Journal of Information Systems*, 18 (2). https://doi.org/10.3127/ajis.v18i2.849.

Cutts, Q., Cutts, E., Draper, S., O'Donnell, P., and Saffrey, P. (2010), 'Manipulating Mindset to Positively Influence Introductory Programming Performance', in *SIGCSE '10: Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 431–35, New York: Association for Computing Machinery. https://doi.org/10.1145/1734263.1734409.

Dasgupta, N., Scircle, M. M., and Hunsinger, M. (2015), 'Female Peers in Small Work Groups Enhance Women's Motivation, Verbal Participation, and Career Aspirations in Engineering', *Proceedings of the National Academy of Sciences*, 112 (16): 4988–93. https://doi.org/10.1073/pnas.1422822112.

Delyser, L. A., Goode, J., Guzdial, M., Kafai, Y., and Yadav, A. (2018), *Priming the Computer Science Teacher Pump: Integrating Computer Science Education into Schools of Education*. New York: CSforALL.

Denner, J. (2011), 'What Predicts Middle School Girls' Interest in Computing?' *Science and Technology*, 2 (1): 17.

Denner, J., Martinez, J., and Thiry, H. (2016), 'Strategies for Engaging Hispanic/Latino Youth in the US in Computer Science', in Y. Rankin and J. Thomas (eds), *Moving Students of Color from Consumers to Producers of Technology*, Advances in Educational Marketing, Administration, and Leadership, Hershey, PA: IGI Global. https://doi.org/10.4018/978-1-5225-2005-4.

Denner, J., Martinez, J., Thiry, H., and Adams, J. (2015), 'Computer Science and Fairness: Integrating a Social Justice Perspective into an After School Program', *Science Education and Civic Engagement: An International Journal*, 6 (2): 41–54.

Denner, J., Werner, L., Campe, S., and Ortiz, E. (2014), 'Pair Programming: Under What Conditions Is It Advantageous for Middle School Students?' *Journal of Research on Technology in Education*, 46 (3): 277–96. https://doi.org/10.1080/15391523.2014.888272.

Duran, R., Hawthorne, E. K., Sabin, M., Tang, C., Weiss, M. A. and Zweben, S. H. (2021), 'Retention in 2017–18 Higher Education Computing Programs in the United States', *ACM Inroads*, 12 (2): 18–28. https://doi.org/10.1145/3448356.

Eccles, L. (2007), *Gender Differences in Teacher-Student Interactions, Attitudes and Achievement in Middle School Science*, Western Australia: Science and Mathematics Education Centre, Curtin University of Technology.

Eccles, J. S., Wigfield, A., and Schiefele, U. (1998). 'Motivation to Succeed', *Handbook of Child Psychology: Social, Emotional, and Personality Development*, 5th ed., vol. 3, 1017–95, New York: Wiley.

Eglash, R., Gilbert, J. E., Taylor, V. and Geier, S. R. (2013), 'Culturally Responsive Computing in Urban, After-School Contexts: Two Approaches', *Urban Education*, 48 (5): 629–56. https://doi.org/10.1177/0042085913499211.

Else-Quest, N. M., Mineo, C. C., and Higgins, A. (2013), 'Math and Science Attitudes and Achievement at the Intersection of Gender and Ethnicity', *Psychology of Women Quarterly*, 37 (3): 293–309. https://doi.org/10.1177/0361684313480694.

Erete, S., Thomas, K., Nacu, D., Dickinson, J., Thompson, N. and Pinkard, N. (2021), 'Applying a Transformative Justice Approach to Encourage the Participation of Black and Latina Girls in Computing', *ACM Transactions on Computing Education*, 21 (4): 1–24. https://doi.org/10.1145/3451345.

Ericson, B. (2021), 'A Deeper Look at Women of Color (Black, Hispanic/Latinx, and Native American) and AP CS in 2020', CSforALL. https://cs4all.home.blog/.

Fernandez, J., and Wilder, J. (2020), 'TECHNOLOchicas: A Critical Intersectional Approach Shaping the Color of Our Future', *Communications of the ACM*, 63 (8): 18–21. https://doi.org/10.1145/3408052.

Fletcher, C. L., and Warner, J. R. (2021), 'CAPE: A Framework for Assessing Equity Throughout the Computer Science Education Ecosystem', *Communications of the ACM*, 64 (2): 23–5. https://doi.org/10.1145/3442373.

Frieze, C., and Quesenberry, J. L. eds. (2019), *Cracking the Digital Ceiling: Women in Computing around the World*, Cambridge: Cambridge University Press.

Gallup Inc. (2021), Developing Careers of the Future: A Study of Student Access to, and Interest in, Computer Science. https://www.gallup.com/analytics/354740/study-of-student-interest-in-computer-science.aspx.

Goode, J., Ivey, A., Johnson, S. R., Ryoo, J. J. and Ong, C. (2021), 'Rac(e)Ing to Computer Science for All: How Teachers Talk and Learn about Equity in Professional Development', *Computer Science Education*, 31 (3): 374–99. https://doi.org/10.1080/08993408.2020.1804772.

Google Inc. and Gallup Inc. (2015), 'Images of Computer Science: Perceptions among Students, Parents, and Educators in the U.S.' https://goo.gl/F3SSWH.

Google Inc. and Gallup Inc. (2016), 'Diversity Gaps in Computer Science: Exploring the Underrepresentation of Girls, Blacks and Hispanics'. https://services.google.com/fh/files/misc/diversity-gaps-in-computer-science-report.pdf.

Guzdial, M. (2013), 'Exploring Hypotheses about Media Computation'. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research*, 19–26, New York: Association for Computing Machinery. https://doi.org/10.1145/2493394.2493397.

Hansen, A. K., Dwyer, H. A., Iveland, A., Talesfore, M., Wright, L., Harlow, D. B. and Franklin, D. (2017), 'Assessing Children's Understanding of the Work of Computer Scientists: The Draw-a-Computer-Scientist Test', in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 279–84, New York: Association for Computing Machinery. https://doi.org/10.1145/3017680.3017769.

Hansen, A. K., Hansen, E. R., Dwyer, H. A., Harlow, D. B. and Franklin, D. (2016), 'Differentiating for Diversity: Using Universal Design for Learning in Elementary Computer Science Education', in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 376–81, New York: Association for Computing Machinery. https://doi.org/10.1145/2839509.2844570.

Happe, L., Buhnova, G., Koziolek, A. and Wagner, I. (2021), 'Effective Measures to Foster Girls' Interest in Secondary Computer Science Education: A Literature Review', *Education and Information Technologies*, 26 (3): 2811–29. https://doi.org/10.1007/s10639-020-10379-x.

HESA (2020, January 16), 'HE Student Enrolments by Subject Area, Sex, First Year Marker, Level of Study, Mode of Study and Country of HE Provider'. https://www.hesa.ac.uk/data-and-analysis/sb255/figure-13.

Hong, H., Wang, J., Ravitz, J., and Fong, M. L. (2015), 'Gender Differences in High School Students' Decisions to Study Computer Science and Related Fields', in *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 689, Kansas City, MI: ACM. https://doi.org/10.1145/2676723.2691920.

Horizon Media (2015), Horizon Media Study Reveals Americans Prioritize STEM Subjects over the Arts; Science Is "Cool", Coding Is New Literacy, *PR Newswire*. https://www.prnewswire.com/news-releases/horizon-media-study-reveals-americans-prioritize-stem-subjects-over-the-arts-science-is-cool-coding-is-new-literacy-300154137.html.

Hubwieser, P., Armoni, M., and Giannakos, M.N. (2015), 'How to Implement Rigorous Computer Science Education in K–12 Schools? Some Answers and Many Questions', *ACM Transactions on Computing Education*, 15 (2): 1–12. https://doi.org/10.1145/2729983.

Huyer, S. (2019), 'A Global Perspective on Women in Information Technology: Perspectives from the "UNESCO Science Report 2015: Towards 2030"', in C. Frieze and J. L. Quesenberry (eds), *Cracking the Digital Ceiling: Women in Computing around the World*, 1st ed., 46–60, Cambridge, UK: Cambridge University Press. https://doi.org/10.1017/9781108609081.003.

Jayathirtha, G., and Kafai, Y. B. (2019), 'Electronic Textiles in Computer Science Education: A Synthesis of Efforts to Broaden Participation, Increase Interest, and Deepen Learning', in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 713–19, Minneapolis, MN: ACM. https://doi.org/10.1145/3287324.3287343.

Jenson, J., Dahya, N., and Fisher, S. (2014), 'Valuing Production Values: A "Do It Yourself" Media Production Club', *Learning, Media and Technology*, 39 (2): 215–28. https://doi.org/10.1080/17439884.2013.799486.

Kapor Center (2021), Culturally Responsive-Sustaining Computer Science Education: A Framework. https://www.kaporcenter.org/equitablecs/.

Khenner, E., and Semakin, I. (2014), 'School Subject Informatics (Computer Science) in Russia: Educational Relevant Areas', *ACM Transactions on Computing Education*, 14 (2): 1–10. https://doi.org/10.1145/2602489.

Koshy, S., Hinton, L., Cruz, L., Scott, A., and Flapan, J. (2021), 'The California Computer Science Access Report', Kapor Center.

Koshy, S., Martin, A., Hinton, L., Scott, A., Twarek, B., and Davis, K. (2021), 'The Computer Science Teacher Landscape: Results of a Nationwide Teacher Survey', Kapor Center. https://www.kaporcenter.org/wp-content/uploads/2021/05/KC21002_-CS-Teacher-Survey-Report_final.pdf.

Kvasny, L., Trauth, E. M., and Morgan, A. J. (2009), 'Power Relations in IT Education and Work: The Intersectionality of Gender, Race, and Class', *Journal of Information, Communication and Ethics in Society*, 7 (2/3): 96–118. https://doi.org/10.1108/14779960910955828.

Lachney, M., Ryoo, J. J., and Santo, R. (2021), 'Introduction to the Special Section on Justice-Centered Computing Education, Part 1', *ACM Transactions on Computing Education*, 21 (4): 1–15. https://doi.org/10.1145/3477981.

Ladner, R. E., and Israel, M. (2016), 'Broadening Participation "for All" in "Computer Science for All" Seeking to Expand Inclusiveness in Computer Science Education', *Communications of the ACM*, 59 (9): 26–8. https://doi.org/10.1145/2971329.

LaPensée, B. A., and Lewis, J. E. (2011), 'Skins: Designing Games with First Nations Youth', *Journal of Game Design & Development Education*, 1 (1), 63–75.

Liben, L. S., and Coyle, E. F. (2014), 'Developmental Interventions to Address the STEM Gender Gap: Exploring Intended and Unintended Consequences', *Advances in Child Development and Behavior*, 47: 77–115. https://doi.org/10.1016/bs.acdb.2014.06.001.

Madkins, T. C., Howard, N. R., and Freed, N. (2020), 'Engaging Equity Pedagogies in Computer Science Learning Environments', *Journal of Computer Science Integration*, 3 (2): 1. https://doi.org/10.26716/jcsi.2020.03.2.1.

Margolis, J., Estrella, R., Goode, J., Jellison, J., and Nao, K. (2008), *Stuck in the Shallow End: Education, Race, and Computing*, Cambridge, MA: MIT Press.

Master, A., Cheryan, S., and Meltzoff, A. N. (2016), 'Computing Whether She Belongs: Stereotypes Undermine Girls' Interest and Sense of Belonging in Computer Science', *Journal of Educational Psychology*, 108 (3): 424–37. https://doi.org/10.1037/edu0000061.

Master, A., Meltzoff, A. N., and Cheryan, S. (2021), 'Gender Stereotypes about Interests Start Early and Cause Gender Disparities in Computer Science and Engineering', *Proceedings of the National Academy of Sciences*, 118 (48): e2100030118. https://doi.org/10.1073/pnas.2100030118.

Muenks, K., Peterson, E. G., Green, A. E., Kolvoord, R. A., and Uttal, D. H. (2020), 'Parents' Beliefs about High School Students' Spatial Abilities: Gender Differences and Associations with Parent Encouragement to Pursue a STEM Career and Students' STEM Career Intentions', *Sex Roles*, 82 (9–10): 570–83. https://doi.org/10.1007/s11199-019-01072-6.

National Center for Education Statistics (2019), Bachelor's, Master's, and Doctor's Degrees Conferred by Postsecondary Institutions, by Sex of Student and Discipline Division: 2017-18. https://nces.ed.gov/programs/digest/d19/tables/dt19_318.30.asp.

Othman, M., and Latih, R. (2006), 'Women in Computer Science: No Shortage Here!' *Communications of the ACM*, 49 (3): 111–14. https://doi.org/10.1145/1118178.1118185.

Peppler, K., Santo, R., Gresalfi, M., and Tekinbas, S. (2014), *Script Changers: Digital Storytelling with Scratch*. Cambridge, MA: MIT Press.

Pinkard, N., Martin, C. K., and Erete, S. (2020), 'Equitable Approaches: Opportunities for Computational Thinking with Emphasis on Creative Production and Connections to Community', *Interactive Learning Environments*, 28 (3): 347–61. https://doi.org/10.1080/10494820.2019.1636070.

Raman, R., Venkatasubramanian, S., Achuthan, K., and Nedungadi, P. (2015), 'Computer Science (CS) Education in Indian Schools: Situation Analysis Using Darmstadt Model', *ACM Transactions on Computing Education*, 15 (2): 1–36. https://doi.org/10.1145/2716325.

Repenning, A., Lamprou, A., Petralito, S., and Basawapatna, A. (2019), 'Making Computer Science Education Mandatory: Exploring a Demographic Shift in Switzerland', in Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, 422–28, Aberdeen, Scotland: ACM. https://doi.org/10.1145/3304221.3319758.

Scott, K. A., Sheridan, K. M., and Clark, K. (2015), 'Culturally Responsive Computing: A Theory Revisited', *Learning, Media and Technology*, 40 (4): 412–36. https://doi.org/10.1080/17439884.2014.924966.

Scott, K. A., and White, M. A. (2013), 'COMPUGIRLS' Standpoint: Culturally Responsive Computing and Its Effect on Girls of Color', *Urban Education*, 48 (5): 657–81. https://doi.org/10.1177/0042085913491219.

Shaw, M. S., Fields, D. A., and Kafai, Y. B. (2020), 'Leveraging Local Resources and Contexts for Inclusive Computer Science Classrooms: Reflections from Experienced High School Teachers Implementing Electronic Textiles', *Computer Science Education*, 30 (3): 313–36. https://doi.org/10.1080/08993408.2020.1805283.

Sien, V. Y., Mui, G. Y., Tee, E. Y. J., and Singh, D. (2014), 'Perceptions of Malaysian Female School Children towards Higher Education in Information Technology'. In *Proceedings of the 52nd ACM Conference on Computers and People Research – SIGSIM-CPR '14*, 97–104, Singapore: ACM. https://doi.org/10.1145/2599990.2600007.

Stefik, A., Ladner, R. E., Allee, W., and Mealin, S. (2019), 'Computer Science Principles for Teachers of Blind and Visually Impaired Students'. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 766–72, Minneapolis, MN: ACM. https://doi.org/10.1145/3287324.3287453.

Vakil, S. (2014), 'A Critical Pedagogy Approach for Engaging Urban Youth in Mobile App Development in an After-School Program', *Equity & Excellence in Education*, 47 (1): 31–45. https://doi.org/10.1080/10665684.2014.866869.

Wang, L. C., and Chen, M. P. (2011), 'The Relationships of Social Economic Status and Learners' Motivation and Performance in Learning from a Game-Design Project', in *Proceedings of the 2011 IEEE 11th International Conference on Advanced Learning Technologies*, 524–28, Athens, GA: IEEE. https://doi.org/10.1109/ICALT.2011.161.

Warner, J. R., Childs, J., Fletcher, C. L., Martin, N. D., and Kennedy, M. (2021), 'Quantifying Disparities in Computing Education: Access, Participation, and Intersectionality', in Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, 619–25, New York: Virtual Event. https://doi.org/10.1145/3408877.3432392.

Wille, S., Century, J., and Pike, M. (2017), 'Exploratory Research to Expand Opportunities in Computer Science for Students with Learning Differences', *Computing in Science & Engineering*, 19 (3): 40–50. https://doi.org/10.1109/MCSE.2017.43.

Zweben, S. (2019), 'Enrollment and Retention in U.S. Computer Science Bachelor's Programs in 2016-17', *ACM Inroads*, 10 (4): 47–59. https://doi.org/10.1145/3366690.

9

Engaging Culturally Relevant and Responsive Pedagogies in Computer Science Classrooms

Tia C. Madkins and Nicol R. Howard

Chapter outline

Chapter synopsis

In this chapter, we provide a general understanding of the current state of the field related to computer science (CS) education and advocate for using equity pedagogies (e.g. culturally relevant pedagogy, culturally responsive computing) through a justice-oriented lens as an essential practice in CS classrooms. We believe that educators and researchers need to understand what it means to use a justice-oriented equity lens in CS teaching and learning and how to engage equity pedagogies in CS classrooms. As such, we provide an overview of a justice-oriented approach to CS education and offer guidance on the distinctions between commonly used equity pedagogies and how educators might use these teaching practices.

9.1 Introduction: Advocating for equity in computer science education

Over the years there has been an increasing interest in Key Stage 1–5 (UK and similar systems) and K–12 (US) CS education as well as efforts to define its goals and questions about how best to reach them (Grover and Pea, 2013). Computer scientist Jeanette Wing's (2006) call to action helped the discussion coalesce around an organizing principle: computational thinking (Caeli and Yadav, 2020; Papert, 1980 [1993]). Connecting this term to twenty-first-century challenges, Wing (2006: 35) argued that beyond just using computer programs or learning to code, the ability to 'think like a computer scientist' is an essential skill for all learners. These conversations spurred initiatives to assess and improve CS education. In a report sponsored by the Association for Computing Machinery (ACM) and the Computer Science Teachers Association (CSTA), Wilson and colleagues (2010) identified three gaps in CS education: a lack of cohesive standards, discrepancies between states and a gap in learners' access and participation in CS courses by racialized, gendered and classed identities. Policymakers and organizations launched initiatives to address these gaps and to begin advocating for equity in CS. For example, in the United States, the Obama administration announced the Computer Science for All Initiative in January 2016, allocating 4 billion dollars to expand access to CS (Smith, 2016) and prompting the founding of the CS for All Consortium. Additionally, organizations like ACM, Code.org, CSTA and the International Society for Technology in Education (ISTE) developed standards and frameworks to define CS education, culminating in the K–12 Computer Science Framework, CSTA standards and ISTE standards.

Importantly, each set of the ISTE standards addresses equity related to learners' access to learning opportunities and achievement in inclusive classroom cultures. For example, the ISTE Computational Thinking Competencies include an Equity Leader strand to provide a framework for goal setting for both teaching and learning. The Equity Leader strand explicitly calls for CS educators to counter stereotypes that exclude any learners from CS opportunities and sets the expectation that educators choose culturally relevant learning activities (ISTE, 2016/17). The CSTA Standards also include an Equity and Inclusion domain addressing a set of CS practices to support teachers in their equity-focused vision (CSTA, 2020).

Key concept: Minoritized learners Minoritized learners

Instead of using *students of colour*, we use the term *minoritized learners* in our research and practice (Madkins and Howard, 2021; Madkins and Morton, 2021). This term highlights the power dynamics and racial hierarchies influencing communities who are in the global majority (Lim, 2020) and remain minoritized in dominant narratives about these children (e.g. minority students, racial minorities). We also use the

term *learners* in our research and practice, rather than *students*, to push back against traditional narratives about children using white, middle-class norms (Madkins and Howard, 2021). This signals that all children are learning regardless of where they are – in or out of school (Madkins and Morton, 2021).

Equity

In education, *equity* is typically defined as *equality* (Gutiérrez, 2008), focusing on fairness and equal access to inputs, like resources (i.e. technology, textbooks, classroom spaces), advanced course offerings, highly qualified teachers and standards-based instruction. Others define equity as related to achievement by highlighting disparate achievement outcomes, like test scores or degree attainment (e.g. Obama CS for All Initiative; NAEP, 2016). In contrast, scholars who have critical stances grounded in social justice define equity by acknowledging minoritized learners' full humanity and viewing their cultural and linguistic practices as resources for learning rather than deficits (i.e. asset-based rather than deficit-oriented approaches). Working towards equity means supporting minoritized learners in the following: (1) engaging in meaningful and rigorous instruction; (2) grappling with and challenging systemic racism, power and oppression and (3) using CS knowledge to empower themselves and their communities. Thus, *equity* is defined as *intentionally* facilitating justice-oriented learning experiences for minoritized learners. This requires viewing teaching and learning as *inseparable* from pursuing justice while attending to learners' access to rigorous instruction and equitable outcomes.

Justice-oriented equity lens

A justice-oriented approach requires three components: (1) prioritizing asset- or strengths-based approaches that centre learners, families and communities; (2) using an equity lens that moves beyond access and achievement frames and instead centres social justice and (3) empowering learners to use CS knowledge for transformation. Ultimately, a justice-oriented equity lens pushes educators to think beyond solving disparate outcomes related to achievement in CS and encourages the consideration of equity-focused teaching practices to support learners in dignity-cultivating learning experiences (Madkins and Howard, 2021).

9.2 Why equity pedagogies in computer science education?

Minoritized learners have historically been marginalized in classrooms and often experience gaps in access to rigorous learning opportunities (i.e. opportunity gaps; Williams, Carter

and Reardon, 2014). Educational research is rife with descriptions of minoritized learners' disparate access to rigorous instruction and achievement outcomes (e.g. Chakrabarti, Carter and Kendi, 2019), especially in CS education. This includes learners' access to technology, advanced course offerings and highly qualified teachers (CSforAll, 2020; Howard and Howard, 2020; Madkins et al., 2019). There are several pedagogical approaches teachers might use to better support minoritized learners and decrease opportunity gaps, collectively known as *equity pedagogies*. These are asset-based teaching practices supporting minoritized learners' outcomes and further developing their potential to become social change agents (Banks and Banks, 1995). Thus, using any equity pedagogy with fidelity *requires* teachers to simultaneously focus on content learning *and* critical consciousness development (i.e. one's familiarity with, recognition of and desire to act upon inequities within a sociohistorical and sociopolitical context (Freire, [1970] 2005)).

Using equity pedagogies can support minoritized children's learning outcomes (i.e. conceptual knowledge development, achievement and identity development) across all content areas, especially CS (Madkins et al., 2019). Though it can be difficult to connect CS course content to social justice issues, educators with justice-oriented approaches to CS education advocate for equity pedagogies within K–12 STEM and CS classrooms. It is incredibly important to address long-standing issues of systemic racism, power and exclusionary practices in CS education. In doing so, we work towards creating and sustaining more just and equitable educational futures for all learners, especially minoritized learners (Bang, 2020).

Educators, scholar activists and others have also called out racism, sexism and anti-Blackness and/or made commitments to addressing equity within K–12 and post-secondary CS education (e.g. Benjamin, 2019; Goode et al., 2020; Guzdial, 2020; Morales-Doyle et al., 2020; Payton et al., 2020; Sherriff et al., 2020; White, 2017). To support researchers and educators in engaging justice-oriented CS education, we outline four theoretical frameworks guiding commonly used equity pedagogies in K–12 CS classrooms.

9.3 Equity pedagogies: Guiding frameworks

Here, we provide an overview of four theoretical frameworks that undergird some of the equity-focused teaching practices teachers might use in CS classrooms: *culturally relevant pedagogy* (CRP; Ladson-Billings, 2006), *culturally responsive teaching* (CRT; Gay, 2000), *culturally sustaining pedagogy* (CSP; Paris and Alim,2017) and *culturally responsive computing* (CRC; Scott and White, 2013, Scott, Sheridan and Clark, 2015). There are other frameworks that may contribute to the understanding and implementation of facilitating equitable CS learning opportunities for minoritized learners. For example, Lee and Soep (2016: 484) introduced *critical computational literacy*, connecting Wing's (2006) computational thinking to critical literacy ('observing, analysing, and deconstructing' oppressive systems and inequalities). However, in this chapter, we briefly highlight the most prominent frameworks (see Table 9.1) in classroom practice based on our review of current research (for comprehensive reviews, see Aronson and Laughter, 2016; Morales-Chicas et al., 2019; Vakil, 2018).

Table 9.1 Differentiating equity-focused theoretical frameworks

Theoretical Framework	Overview
Culturally relevant pedagogy (CRP) Ladson-Billings (2006)	Three tenets include educators' intentional mindset and explicit actions related to the following: • Academic excellence (explicit high expectations for learning and achievement); • Cultural competence (using learners' cultural practices for learning and breaking down dominant culture in schooling); and • Critical/sociopolitical consciousness development (awareness of and challenging of sociopolitical forces influencing our world).
Culturally responsive teaching (CRT) Gay (2000) Hammond (2015a, 2015b)	The five essential elements of culturally responsive teaching include the following: • Cultural diversity knowledge base; • Culturally relevant curricula; • Cultural care and learning communities; • Cross-cultural communications; and • Cultural congruity.
Culturally sustaining pedagogy (CSP) Paris (2012) Paris and Alim (2017) Also see Chapter 10 by Israel et al.	Focus on maintaining and cultivating linguistic and cultural pluralism, in opposition to existing educational contexts historically structured to ignore and marginalize learners' cultural and linguistic assets.
Culturally responsive computing (CRC) Scott and White, 2013; Scott, Sheridan and Clark, 2015)	Draws and builds upon principles of culturally relevant teaching to honour learners' backgrounds, life experiences and interests to make meaningful connections to computing topics.

Source: Madkins, Howard and Freed (JCSI, 2020)

Culturally relevant pedagogy

Using CRP in the CS classroom demonstrates a teacher's explicit acknowledgement that they value all learners and expect them to excel. CRP principles include (a) high expectations for all learners, (b) development and maintenance of cultural competence and learners' development of cultural capital to succeed in the dominant culture and (c) planning and facilitating learning experiences to further develop learners' critical consciousness (Ladson-Billings, 2006). Teachers must give equal attention to *each* component to implement CRP with fidelity. Teachers engaging CRP also show their learners they genuinely care for them, reject deficit-model thinking about minoritized learners and view learners' cultural practices as resources for learning (Ladson-Billings, 2006).

Culturally responsive teaching

Geneva Gay (2000) defines CRT as situating the lived cultural experiences, characteristics and perspectives of ethnically diverse learners as a primary channel to inform effective teaching. The five essential elements of CRT include the following: (1) developing a cultural diversity knowledge

base; (2) designing culturally relevant curricula; (3) demonstrating cultural care and building learning communities; (4) effective cross-cultural communications and (5) cultural congruity. Prior to delivering instruction, teachers create a classroom climate of cultural care, in partnership with their learners, that is conducive to learning while fostering communal classrooms. Teachers also prioritize understanding their learners' communication styles to avoid violating cultural values and to achieve effective cross-cultural communication in classrooms. Teachers also establish cultural congruity by taking everything they know about their learners' cultural diversity (and how their learners learn) to enhance their instructional delivery.

Zaretta Hammond (2015a, b) builds on the five essential elements provided by Gay (2000) through her work on CRT and the brain through four interdependent practices: *awareness*, *learning partnerships*, *information processing* and *community building*. Hammond (2015a, b) asserts that culturally responsive teachers who have *awareness* develop their own sociopolitical consciousness. Building a *learning partnership* with learners in the instructional process requires teachers to build trust and move away from didactic approaches to reimagine the relationship as a partnership for education with authentic opportunities to make cultural connections (Hammond, 2015a). The culturally responsive teacher works to create an environment where minoritized learners have opportunities to engage in *information processing* 'in a manner congruent with how the brain learns' (Hammond, 2015ba: 172) and take risks because they feel they belong and are fully supported. Finally, a culturally responsive teacher engages in *community building*, whereby they establish a safe and intellectual environment for learners so they feel supported by one another.

Culturally sustaining pedagogy

Building upon CRP (Ladson-Billings, 2006), Paris (2012) and colleague (Paris and Alim, 2017) conceptualized CSP. This approach promotes equity by engaging minoritized learners in experiences that sustain cultural pluralism, where educators recognize and build upon the varied and dynamic nature of learners' repertoires of practice (Gutiérrez and Rogoff, 2003). Moreover, educators must reject the idea that learning to navigate the dominant culture (i.e. white, middle-class norms) is the *only* way for learners to be successful. As such, youth have opportunities to learn in a manner that sustain their cultural ways of being and shape their access to power in oppressive systems (Paris and Alim,2017). By utilizing CSP, educators and learners both hope to understand, explore and critique how racialized, ethnic and/or cultural experiences (e.g. what it means to be Latinx or Black in a particular context) are rooted in both enduring *and* shifting practices (also see Chapter 8 by Denner and Campe).

Culturally responsive computing

Kimberly Scott and colleagues (Scott and White, 2013; Scott et al., 2015) introduced CRC to make connections between culturally relevant teaching practices and CS. These principles support efforts focusing on increasing learners' access to computing and diversity in the tech fields but with

a focus on the inextricable link between STEM and computing content and social justice (Scott, Sheridan and Clark, 2015).

CRC embeds sociocultural relevance at all levels of the learning experience, from the selection of tools and the actual classroom to applications outside of the classroom. Thus, learners are supported to creatively engage with technology in a meaningful context with attention to their cultural and personal interests while addressing intersectional identities that impact their experiences with technology. CRC calls for situating technology ideas within a sociopolitical context and giving learners opportunities to critique and explore issues they find relevant. Learners should feel supported in their learning, identity development and expression as they become creative innovators with technology, able to repurpose computing tools towards their own goals. When using CRC, educators redefine *success* of computing programs by asking 'who creates, for whom and to what ends?' rather than gauging success by a list of CS topics covered.

In summary, CRC principles include the following: (1) developing technologies that reflect and respond to minoritized learners' identities; (2) critiquing and addressing sociopolitical issues within computing and society more broadly and (3) facilitating computing learning experiences that are grounded in and built upon a rigorous curriculum *and* minoritized learners' identities (e.g. academic, racialized, gendered) and cultural and linguistic practices (Eglash et al., 2013; Scott, Sheridan, and Clark, 2015).

9.4 Equity pedagogies in practice

Recent scholarship has shown how using equity pedagogies in K–12 CS classrooms can positively influence student outcomes, such as achievement, belonging and interest (Ryoo et al., 2013; Scott and White, 2013; Vakil, 2014). Other scholars have found the use of culturally relevant and responsive teaching to support learners in making connections between social justice issues, CS course content and personal relevance (Madkins et al., 2019; Scott, Sheridan and Clark, 2015).

Scott and White (2013) utilized CRC principles in their curriculum for COMPUGIRLS, a two-year program for girls, who were mostly from Black and Latinx communities. The use of CRC practices facilitated girls' increased tech knowledge, identification with and interest in CS, agency and critical consciousness development. Vakil (2014) highlights the importance of connecting app programming to critical consciousness development – a key component of equity-focused pedagogies. Learners expressed that having opportunities to make connections to social justice issues contributed to their increased interest, engagement and participation in CS (Vakil, 2014).

Exploring computer science (ECS), a popular curriculum and course in high schools, was developed in 2008 to democratize CS learning through culturally relevant curricula (Goode and Margolis, 2011; Ryoo et al., 2013). ECS is grounded in learners' enquiry-based explorations of CS concepts and issues of equity and is aligned with CRC principles (Ryoo, 2019). Across ECS classrooms, learners have been supported in developing critical consciousness, seeing the relevance of CS to their lives and learning CS concepts (Goode and Margolis, 2011; Ryoo, 2019; Ryoo et al., 2013).

Example: Equity-focused teaching practices in CS

Here, we share practical examples to demonstrate how educators might implement equity-focused teaching in CS classrooms.

Table 9.2 Engaging equity-focused teaching practices

Identity development	• Are aware of and address power dynamics that impact classroom culture and learners' experience • Are aware that intersectional identities can impact learners' experiences with technology in different ways and on different levels (might include: how the effects of technology impact different communities and populations differently, identity impacts on access to technology/technology learning, identity impacts on ability to see oneself as a technological change agent) • Support identity development and expression through computing tools • Challenge stereotypes about who belongs in computing by addressing representation, showing learners examples of diverse creators and creating space for discussion • Encourage peer pedagogy, where learners teach and learn from each other via feedback and sharing ideas/completed work (for a full explanation, see Ching and Kafai, 2008, or Fields et al., 2018) • Support learner–learner friendships
Personal and sociopolitical relevance of technology	• Situate technology ideas within their sociopolitical context and give learners opportunities to critique and explore issues that are relevant to them • Create CS experiences connected to learners' knowledge, interests and life experiences, broadening ideas of where CS thinking can be applied • Value learners' strengths and outside knowledge and create opportunities for them to showcase these
Positioning learners as creative agents/ change agents	• Empower learners to become creative innovators with technology, able to repurpose technology towards their own goals • Build enough flexibility into both the tools and the activities to leave room for learners to pursue goals that are meaningful to them • Legitimize learners' expertise through creating opportunities to share their work with the broader community and/or making their work visible in a shared space

Sources: Ashcraft et al. (2017); Babbitt et al. (2012); Fields et al. (2018); Keune et al. (2019); Pinkard et al. (2017); Ryoo (2019); Scott, Sheridan, and Clark (2015); Vakil (2018).

9.5 Conclusion

When providing the background and context on the state of the field of CS, we noted the renewed interest in CS education over the past fifteen years. From the Obama administration announcement of the Computer Science for All initiative to the collective work informing the Computer Science Frameworks and subsequent standards, direct and indirect efforts are underway to improve the

field and broaden participation in CS. This increased interest in broadening participation also requires relevant guidance for situating equity pedagogies as an essential practice for our work with minoritized learners. Both the ISTE and CSTA standards for educators address equity related to access and achievement. These sets of standards include words like equity, inclusion and culturally relevant learning, demonstrating a recognition that there is still a need for the purposeful application of instructional practices. We commend the collective work of CTSA, ISTE and other organizations. However, as critical scholars of STEM and CS education, we would be remiss if we did not acknowledge that what is written in a set of standards is often either not followed or explicitly followed absent of the critical consciousness or rationale required to effectively implement equity pedagogies.

In this chapter, we provided a rationale for using equity pedagogies in CS and outlined practical examples from research to illustrate how practitioners might use equity pedagogies in their classrooms. We offered background and insights related to four prominent theoretical frameworks grounded in asset-based approaches to supporting minoritized learners as evidenced by research and practice. Although we differentiate these equity-focused theoretical frameworks, the application of any of these frameworks positions educators to continually work towards centring equity.

Effectively engaging equity pedagogies requires educators to support minoritized learners by: (1) constructing culturally relevant and rigorous CS curricula; (2) implementing the curricula in a meaningful manner; (3) challenging systemic racism, power and oppression; (4) guiding learners in grappling with and challenging these same systems and (5) encouraging them to use CS to empower themselves and their communities. When educators engage equity pedagogies, learners are supported in their identity development, understand the personal and sociopolitical relevance of technology and are positioned as change agents.

CS innovations remain an important factor in the growth and development of economies around the world. Minoritized learners continue to contend with the denial of access to advanced and rigorous CS (and other STEM) courses. Much attention has been given to broadening participation for minoritized learners, and there have been notable advances in CS educational research and practitioner communities related to equity-focused teaching. Yet, there is more work to do to further support learners with the integration of CS knowledge in K–12 classrooms. Such an integration can occur by prioritizing equity pedagogies in CS classrooms to effectively prepare learners for entry into CS-related fields. It is our hope that this chapter uplifts efforts to support minoritized youth in further developing their CS knowledge, seeing the connections between using this knowledge and pursuing their personal interests and using computational thinking to empower themselves and their communities.

In conclusion, we emphasize that equity-focused work is important because we cannot continue to invite learners (and, in turn, their families and communities) into CS education by focusing only on increasing access to CS courses, development of CS knowledge and working towards CS integration. If we, instead, engage in CS teaching and learning with a justice-oriented approach, we are more likely to invite them into a field and learning experience that they will welcome and appreciate. What is most important to remember is that classroom-based or informal educators, teacher educators, district personnel, families and community members are all working together with common goals and with a purpose – supporting all learners to be successful in CS education.

We know that content and context matter, so the ways we implement equity-focused teaching practices will look different wherever learning occurs. If we hold each other accountable to actually engage equity-focused teaching, we get better at it over time. It is imperative for educators to be kind to one another and extend grace to self and colleagues as we hold each other accountable to do the difficult work.

Key points

- This chapter notes the importance of working towards equity in support of minoritized learners.
- Four asset-based equity pedagogies (CRP, CRT, CSP and CRC) were highlighted as well as the ways in which they should be taken up in K–12 computer science classrooms.

For further reflection

1. What do you view as the distinctions between the four frameworks we highlighted in this chapter (culturally relevant pedagogy, culturally responsive teaching, culturally sustaining pedagogy and culturally responsive computing)?
2. Think of a lesson you recently taught *or* will teach soon. How could you modify it to centre a social justice issue related to the content or concepts you teach in that lesson?
3. How might you incorporate your learners' lived experiences into a CS lesson?
4. What are the affordances of engaging culturally relevant and responsive pedagogies in CS teaching for learners, their families and the community?
5. Considering the importance of equity-centred pedagogies in CS, in what ways are you engaging these practices? What will you do differently?

References

Aronson, B., and Laughter, J. (2016), 'The Theory and Practice of Culturally Relevant Education: A Synthesis of Research across Content Areas', *Review of Educational Research*, 86 (1): 163–206. doi: 10.3102/0034654315582066.

Ashcraft, C., Eger, E. K., and Scott, K. A. (2017), 'Becoming Technosocial Change Agents: Intersectionality and Culturally Responsive Pedagogies as Vital Resources for Increasing Girls' Participation in Computing', *Anthropology & Education Quarterly*, 48 (3): 233–51. doi: 10.1111/aeq.12197.

Babbitt, B., Lyles, D., and Eglash, R. (2012), 'From Ethnomathematics to Ethnocomputing: Indigenous Algorithms in Traditional Context and Contemporary Simulation', in S. Mukhopadhyay and W-M. Roth (eds), *Alternative Forms of Knowing (in) Mathematics*, 205–19, Boston, MA: Brill Sense.

Bang, M. (2020), 'Learning on the Move toward Just, Sustainable, and Culturally Thriving Futures', *Cognition and Instruction*, 38 (3): 434–44. doi: 10.1080/07370008.2020.1777999.

Banks, C. A. M., and Banks, J. A. (1995), 'Equity Pedagogy: An Essential Component of Multicultural Education', *Theory into Practice*, 34 (3): 151–8. doi: 10.1080/00405849509543674.

Benjamin, R. (2019), *Race after Technology: Abolitionist Tools for the New Jim Code*, Cambridge, UK: Polity.

Caeli, E. N., and Yadav, A. (2020), 'Unplugged Approaches to Computational Thinking: A Historical Perspective', *TechTrends*, 64 (1): 29–36. doi: 10.1007/s11528-019-00410-5.

Chakrabarti, M., Carter, P. L., and Kendi, I. (2019, 9 September). *Part I: Achievement Gap, or Opportunity Gap? What's Stopping Student Success*. WBUR Closing the Achievement Gap Series, Boston, MA: WBUR.

Ching, C. C., and Kafai, Y. B. (2008), 'Peer Pedagogy: Student Collaboration and Reflection in a Learning-through-Design Project', *Teachers College Record*, 110 (12), 2601–32.

Computer Science Teachers Association (CSTA) (2020), CSTA Standards for Computer Science Teachers. https://csteachers.org/teacherstandards.

CSforAll (2020), Equitable CS District Learning Cohort. https://www.csforall.org/projects_and_programs/equitable-cs-district-learning-cohort/.

Eglash, R., Gilbert, J. E., Taylor, V., and Geier, S. R. (2013), 'Culturally Responsive Computing in Urban, After-School Contexts: Two Approaches', *Urban Education*, 48 (5): 629–56. https://doi.org/10.1177/0042085913499211.

Fields, D. A., Kafai, Y., Nakajima, T., Goode, J., and Margolis, J. (2018), 'Putting Making into High School Computer Science Classrooms: Promoting Equity in Teaching and Learning with Electronic Textiles in Exploring Computer Science', *Equity & Excellence in Education*, 51 (1): 21–35. doi: 10.1080/10665684.2018.1436998.

Freire, P. ([1970] 2005), *Pedagogy of the Oppressed*, London: Continuum.

Gay, G. (2000), *Culturally Responsive Teaching: Theory, Research, and Practice*, New York: Teachers College Press.

Goode, J., and Margolis, J. (2011), 'Exploring Computer Science: A Case Study of School Reform', *ACM Transactions on Computing Education*, 11 (2): 1–16. doi: 10.1145/1993069.1993076.

Goode, J., Ivey, A., RunningHawk Johnson, S., Ryoo, J. J., and Ong, C. (2020), 'Rac(e)ing to Computer Science for All: How Teachers Talk and Learn about Equity in Professional Development', *Computer Science Education*, 31 (3): 374–99. https://doi.org/10.1080/08993408.2020.1804772.

Grover, S., and Pea, R. (2013), 'Computational Thinking in K–12: A Review of the State of the Field', *Educational Researcher*, 42 (1): 38–43. https://doi.org/10.3102/0013189X12463051.

Gutiérrez, K. D., and Rogoff, B. (2003), 'Cultural Ways of Learning: Individual Traits or Repertoires of Practice', *Educational Researcher*, 32 (5): 19–25. https://doi.org/10.3102/0013189X032005019.

Gutiérrez, R. (2008), 'Framing Equity: Helping Learners "Play the Game" and "Change the Game"', *Noticias de Todos/TODOS: Mathematics for ALL*, 4 (1): 1–3.

Guzdial, M. (2020, 5 June), CS Teachers, It's (Past) Time to Learn about Race, Blog@CACM. https://cacm.acm.org/blogs/blog-cacm/245408-cs-teachers-its-past-time-to-learn-about-race/fulltext.

Hammond, Z. (2015a), *Culturally Responsive Teaching and the Brain: Promoting Authentic Engagement and Rigor among Culturally and Linguistically Diverse Learners*, Thousand Oaks, CA: Corwin. https://crtandthebrain.com/four-tools-for-interrupting-implicit-bias/.

Hammond, Z. (2015b, 9 April), Four Tools for Interrupting Implicit Bias, *Culturally Responsive Teaching and the Brain*. https://crtandthebrain.com/four-tools-for-interrupting-implicit-bias/.

Howard, R., and Howard, K. E. (2020), *Coding + Math: Strengthen K–5 Math Skills with Computer Science*, Portland, OR: International Society for Technology in Education.

International Society for Technology in Education (ISTE) (2016/17), *The ISTE Standards*, Eugene, OR: International Society for Technology in Education. https://www.iste.org/standards/iste-standards-for-teachers.

Keune, A., Peppler, K. A., and Wohlwend, K. E. (2019), 'Recognition in Makerspaces: Supporting Opportunities for Women to "make" a STEM Career', *Computers in Human Behavior*, 99: 368–80. doi: 10.1016/j.chb.2019.05.013.

Ladson-Billings, G. (2006), '"Yes, but How Do We Do It?" Practicing Culturally Relevant Pedagogy', in J. Landsman and C. W. Lewis (eds), *White Teachers/Diverse Classrooms: A Guide to Building Inclusive Schools, Promoting High Expectations, and Eliminating Racism*, 29–42, Loudoun County, VA: Stylus.

Lee, C. H., and Soep, E. (2016), 'None but Ourselves Can Free Our Minds: Critical Computational Literacy as a Pedagogy of Resistance', *Equity & Excellence in Education*, 49 (4): 480–92. doi: 10.1080/10665684.2016.1227157.

Lim, D. (2020, 9 May), I'm Embracing the Term "People of the Global Majority", Medium.com. https://regenerative.medium.com/im-embracing-the-term-people-of-the-global-majority-abd1c1251241.

Madkins, T. C., and Howard, N. R. (2021), Equity-Focused Teaching in K–12 CS: Strategies for Teachers, Teacher Educators, and Districts, *Raspberry Pi Foundation Seminar Proceedings*, 11–17. https://www.raspberrypi.org/app/uploads/2021/12/Understanding-computing-education-Volume-2-Raspberry-Pi-Foundation-Research-Seminars.pdf.

Madkins, T. C., Howard, N. R., and Freed, N. (2020), 'Engaging Equity Pedagogies in Computer Science Learning Environments', *Journal of Computer Science Integration*, 3 (2): 1–27 (open access). doi: 10.26716/jcsi.2020.03.2.1.

Madkins, T. C., Martin, A., Ryoo, J., Scott, K. A., Goode, J., Scott, A., and McAlear, F. (2019), 'Culturally Relevant Computer Science Pedagogy: From Theory to Practice', in *Proceedings of the* 2019 Research on Equity and Sustained Participation in Engineering, Computing, and Technology, RESPECT-*2019*, 1–4, Minneapolis, MN. doi: 10.1109/RESPECT46404.2019.8985773.

Madkins, T. C., and Morton, K. (2021), 'Disrupting Anti-Blackness with Young Learners in STEM: Strategies for Elementary Science and Math Teacher Education', *Canadian Journal of Science, Mathematics, and Technology Education*, 21 (2): 239–56. https://doi.org/10.1007/s42330-021-00159-1.

Morales-Chicas, J., Castillo, M., Bernal, I., Ramo, P., and Guzman, B. L. (2019), 'Computing with Relevance and Purpose: A Review of Culturally Relevant Education in Computing', *International Journal of Multicultural Education*, 21 (1): 125–55.

Morales-Doyle, D., Vossoughi, S., Vakil, S., and Bang, M. (2020, 19 August), 'In an Era of Pandemic, STEM Education Can't Pretend to Be Apolitical', *Truthout Education and Youth* (op-ed).

National Assessment of Educational Progress (NAEP) (2016), *Scores 2015*. Washington, DC: National Center for Education Statistics. https://nces.ed.gov/nationsreportcard/studies/tbaproject.aspx.

Papert, S. ([1980] 1993), *Mindstorms. Children, Computers, and Powerful Ideas*, New York: Basic Books.

Paris, D. (2012), 'Culturally Sustaining Pedagogy: A Needed Change in Stance, Terminology, and Practice', *Educational Researcher*, 41 (3): 93–7. doi: 10.3102/0013189X12441244.

Paris, D., and Alim, H. S. (eds) (2017), *Culturally Sustaining Pedagogies: Teaching and Learning for Justice in a Changing World*, New York: Teachers College Press.

Payton, J., Barnes, T., Gardner-McCune, C., and Washington, N. (2020), STCBP & RESPECT Statement on Addressing Racism & Injustice. http://stcbp.org/respect-statement-on-addressing-racism-injustice/.

Pinkard, N., Erete, S., Martin, C. K., and McKinney de Royston, M. (2017), 'Digital Youth Divas: Exploring Narrative-Driven Curriculum to Spark Middle School Girls' Interest in Computational Activities', *Journal of the Learning Sciences*, 26: 477–516. doi: 10.1080/10508406.2017.1307199.

Ryoo, J. J. (2019), 'Pedagogy That Supports Computer Science for All', *ACM Transactions on Computing Education*, 19 (4): 1–23. doi: 10.1145/3322210.

Ryoo, J. J., Margolis, J., Lee, C. H., Sandoval, C. D. M., and Goode, J. (2013), 'Democratizing Computer Science Knowledge: Transforming the Face of Computer Science through Public High School Education', *Learning, Media, and Technology*, 28 (2), 161–81. doi: 10.1080/17439884.2013.756514.

Scott, K. A., and White, M. (2013), COMPUGIRLS' Standpoint: Culturally Responsive Computing and Its Effect on Girls of Color', *Urban Education*, 48: 657–81. doi: 10.1177/0042085913491219.

Scott, K. A., Sheridan, K. M., and Clark, K. (2015), 'Culturally Responsive Computing: A Theory Revisited', *Learning, Media and Technology*, 40: 412–36. doi: 10.1080/17439884.2014.924966.

Sherriff, M., Merkle, L., Cutter, P., Monge, A., and Sheard, J. (2020), 'Statement on Equity', Association for Computing Machinery (ACM) Special Interest Group on Computer Science Education (SIGCSE) Technical Symposium and Program Co-Chairs.

Smith, M. (2016, 30 January), 'Computer Science for All Summary', The White House (President Barack Obama).

Vakil, S. (2014), 'A Critical Pedagogy Approach for Engaging Urban Youth in Mobile App Development in an After-School Program', *Equity and Excellence in Education*, 47: 31–45. doi: 10.1080/10665684.2014.866869.

Vakil, S. (2018), 'Ethics, Identity, and Political Vision: Toward a Justice-Centered Approach to Equity in Computer Science Education', *Harvard Educational Review*, 88: 26–52. doi: 10.17763/1943-5045-88.1.26.

White, S. V. (2017, 9 July), Educators, It's Time to #TakeAKnee? Blog post. https://medium.com/identity-education-and-power/educators-its-time-to-takeaknee-bb470d0c996a.

Williams, J., Carter, P. L., and Reardon, S. (2014), 'Researchers on Opportunity Gap, Student Inputs, Outputs', Podcast on bloomberg.com.

Wilson, C., Sudol, L. A., Stephenson, C., and Stehlik, M. (2010), 'Running on Empty: The Failure to Teach K–12 Computer Science in the Digital Age', *Executive Summary*, 1–13. New York: Association for Computing Machinery.

Wing, J. M. (2006), 'Computational Thinking', *Communications of the ACM*, 49 (3): 33–5. https://doi.org/10.1145/1118178.1118215.

10

Increasing Access, Participation and Inclusion within K–12 CS Education through Universal Design for Learning and High Leverage Practices

Maya Israel, Latoya Chandler, Alexis Cobo and Lauren Weisberg

Chapter outline

Chapter synopsis

What do we mean by Computer Science for All? How can we plan computer science (CS) in a manner that proactively includes all learners? In this chapter, we explore strategies for including all learners in K–12 CS education through the lens of inclusive practices that include Universal Design for Learning (UDL) and High Leverage Practices (HLPs). We present the theoretical underpinnings of these two approaches as well as how they connect to K–12 CS education research. Lastly, we also provide guidance about the intersection of UDL with Culturally Responsive and Sustaining Education.

10.1 Introduction: Computer science for all learners

Computer science (CS) education is rapidly becoming part of the core curriculum for young people around the globe. Because of the ubiquitous nature of computation in today's society, there is a growing need to prepare citizens with fundamental computational literacies, both to meet the professional demands of global economies and to fulfill their public roles in society (Grover and Pea, 2013; Tissenbaum, Sheldon and Abelson, 2019; Wing, 2006). Initiatives such as *CS for All* seek to help students develop these essential computational literacies and, in the process, democratize computing by focusing on access and inclusion of all learners.

Although the phrase *CS for All* has been used widely in the CS education community, many such efforts have yet to focus on the inclusion of students with disabilities. In the United States alone, there are approximately 7.3 million students with disabilities (approximately 14 per cent of students) receiving special education services in public K–12 schools under the Individuals with Disabilities Education Act (Irwin et al., 2021). Although more difficult to track, approximately another 2.3 per cent of students in the United States receive accommodations through section 504 of the Rehabilitation Act of 1973, a civil rights law that prohibits discrimination based on disability (Zirkel, 2019). Given these numbers, if we are committed to CS for All, we must increase efforts for the inclusion of students with disabilities in CS education.

Understanding barriers and pathways to inclusion in K–12 CS education

Despite the large number of students with disabilities in K–12 schools, with a few notable exceptions, there has been a surprising silence regarding their inclusion in CS education. This shortage of attention to the inclusion of students with disabilities in K–12 CS education has directly resulted in exclusionary practices including a lack of development and utilization of accessible computational tools, limited professional development in inclusive pedagogical approaches and a lack of advocacy for the rights of students with disabilities to participate in K–12 CS instruction (Israel et al., in press). This phenomenon is not unique to CS education. However, because CS education is now emerging as a new disciplinary area in many schools (Kirby, 2017), there is an opportunity to disrupt this trend early in order to increase access and engagement for all learners.

The small but growing body of literature about the inclusion of students with disabilities in K–12 CS education indicates that when these learners are presented with accessible tools and inclusive pedagogical approaches, they are likely to succeed (e.g. Israel et al., 2020; Snodgrass, Israel and Ladner, 2016). Rather than focusing on 'fixing problems within students', which is a positionality based on a deficit model of disability, many of these studies place inclusive practices as a pathway to participation and learning. Thus, by taking a social view on disability, we flip the narrative towards an emphasis on reducing barriers in the environment that prevent learning (Israel et al., in press).

It is helpful to use an ecological systems theory approach to understanding both barriers and pathways to inclusion within K–12 CS education (Bronfenbrenner, 1977). Understanding the experiences of students with disabilities is complex, not uniform, and is based on the environment in which a student learns. At the microsystem level, the student engages with parents, teachers, peers and others. At the mesosystem level, the student experiences policies within the school and district such as advocacy for (or lack of) accessible materials and school-level scheduling decisions. At the macrosystem level, the student is influenced by educational policies and laws related to CS education mandates as well as disability-specific legislation related to inclusion and participation. At each of these levels, there are barriers and pathways to inclusion that are situated within a complex system that cannot be discussed only at the classroom level (Israel, 2021). Table 10.1 provides examples of both barriers and pathways to inclusion in K–12 CS education.

Key concept: Inclusive mindsets

The term 'inclusive practices' is often used broadly to suggest a set of approaches that increase the participation and belongingness of students with disabilities, thus proactively minimizing their exclusion (Florian and Black-Hawkins, 2011). To enact these practices, teachers must first believe that students with disabilities belong to and can succeed in K–12 CS education. These mindsets assume five equity principles (adapted from Israel, 2021):

1. All learners deserve to be meaningfully included in CS education.
2. All learners can succeed in CS instruction.
3. Learner variability is an asset in the CS classroom.
4. CS instruction must engage all learners.
5. Advocating for CS inclusion challenges barriers and opens pathways to participation.

Understanding and reflecting on these five equity principles can help teachers build an inclusive mindset and provides a starting place for understanding how to apply the instructional frameworks described below.

10.2 Applying inclusive frameworks in K–12 CS education classrooms

In order to achieve K–12 CS instruction that is accessible and inclusive and encourages full participation and belongingness of all learners, teachers must proactively plan for that level of participation. In this section, two approaches are described: Universal Design for Learning (UDL) and high leverage practices (HLPs). UDL is a general framework for proactively planning for inclusion and participation and HLPs are a set of instructional strategies that can be implemented within UDL-based instruction.

Table 10.1 Barriers and pathways to inclusion in K–12 CS education

	Example Barriers	Example Pathways
Macrosystem	Lack of explicit policies about inclusion of students with disabilities accessing CS education	Advocacy at all levels for inclusion and participation of all students (including those with disabilities) in CS education.
	Limited mandates for purchasing of accessible CS tools, materials and curricula	Expectations and accountability by educational leaders about the development of accessible CS tools, materials and curricula.
Mesosystem	Limited professional development for teachers about inclusive pedagogical approaches	Investment of time and resources into teacher professional development focused on inclusion and participation of students with disabilities in K–12 CS education
	CS offered only to students taught in inclusive educational contexts, leaving out students who are not in these settings	CS is available for and encouraged by all K–12 students
	School counsellors advising students with disabilities to not take CS courses	
Microsystem	Assumptions by teachers that students with disabilities cannot succeed in CS	Use of inclusive pedagogies such as Universal Design for Learning and high leverage practices in K–12 CS instruction
	Lack of availability of assistive technologies in classrooms	Advocacy by teachers for the full participation of students with disabilities in CS instructional activities
	Limited use of inclusive pedagogical approaches in CS instruction	

Source: Adapted from Israel (2021).

Universal Design for Learning in CS education

UDL is a pedagogical framework developed by the Center for Applied Special Technology (CAST), a nonprofit research and development organization that promotes the expansion of equitable learning opportunities. Following the passage of the Americans with Disabilities Act in 1990, the universal design movement first gained popularity with architects and designers seeking to make environments and products more accessible to those with disabilities. UDL evolved alongside a trend towards inclusive educational policies designed to increase instructional accessibility for students with disabilities. The UDL framework is rooted in research-based educational practices (Posey, n.d.) and can be used as 'a blueprint for creating flexible goals, methods, materials, and assessment that meet the needs of diverse learners' (Rose, Meyer and Hitchcock, 2005: 3).

Rather than promoting an ideal method of instructional delivery, UDL prioritizes flexibility in instructional methods, materials and learning environments with the goal of removing barriers to learning, designing lessons for academically diverse classrooms and addressing the unique learning needs of students with disabilities (Israel, Lash and Ray, 2017a; 2020). Learners in a UDL classroom rarely perform the same tasks concurrently. Rather, they engage in flexible activities and assessments with a variety of learning materials and strategies designed to empower and motivate them to take control over their learning experiences. UDL was developed alongside advances in the field of cognitive neuroscience such as the identification of key brain networks responsible for learning (CAST, 2018). These networks align with UDL's three guiding principles: multiple means

of engagement (the 'why' of learning), multiple means of representation (the 'what' of learning) and multiple means of action and expression (the 'how' of learning). Applying these guidelines to one's curriculum and pedagogy can result in instruction that is inclusive, accessible and usable (Burgstahler, 2020).

The UDL guidelines are on the CAST website and include various resources designed to help educators with practical application in their curricula. Israel and colleagues (2017) also developed a tool that provides recommendations for integrating UDL into CS and computational thinking curricula (see Table 10.2 for an adapted version). This table includes recommendations for various strategies for designing CS lessons with the three UDL principles. For instance, to ensure that a lesson has multiple means of engagement, options should be provided for recruiting learner interest, sustaining learner effort and persistence and facilitating learners' self-regulation of behaviour. This principle can be accomplished by giving students a choice regarding their projects, software or topics; teaching and encouraging peer collaboration and product sharing; and developing ways for students to self-assess and reflect on their own work and the work of others. To ensure that a lesson has multiple means of representation, information should be delivered with various languages, symbols and modalities to provide students options for how they perceive and comprehend information. This principle can be accomplished by teaching and reviewing content-specific vocabulary, modelling computing using physical representations as well as interactive whiteboards and videos, and providing graphic organizers for students to 'translate' programs into pseudocode. To ensure that a lesson has multiple means of action and expression, options should be provided for learners to engage in physical action, expression and communication, and executive functions. This principle can be accomplished by including CS Unplugged activities that show the physical relationship of abstract computing concepts, providing opportunities to practice computing skills and content through projects that build on prior lessons and guiding students to set goals for long-term projects.

Key concept: UDL and key legislation

In the United States, UDL is endorsed by various K–12 education policy initiatives (CAST, n.d.) including the Every Student Succeeds Act (2015), which requires that assessments be designed with UDL principles in mind and the National Education Technology Plan (Office of Educational Technology, 2017), which urges that technology be 'born accessible' to increase access and inclusion for all learners. In these laws, UDL commonly leverages assistive technologies to improve accessibility of instruction and proactive support learning for students who benefit from accessible instructional materials (Basham et al., 2010; Posey, n.d.). Evaluating tools commonly used in K–12 CS instruction is important for meeting learning goals in an inclusive classroom.

Scholars advocate that a UDL-based instructional approach is crucial for achieving equitable and accessible CS instruction that meets the needs of diverse learners, including those with disabilities (Hansen et al., 2016; Israel, Lash and Ray, 2017a; Israel and Ladner, 2016; Ray et al.,

Table 10.2 UDL in K–12 CS education crosswalk

	Providing Multiple Means of *Engagement*	**Providing Multiple Means of *Representation***	**Providing Multiple Means of *Action and Expression***
	Affective networks: The 'WHY' of learning	Recognition networks: The 'WHAT' of learning	Strategic networks: The 'HOW' of learning
Access	Provide options for *recruiting interest*: Give students choices (choose project, software, topic) Allow students to make projects relevant to culture and age Allow for differences in pacing and length of work sessions	Provide options for *perception*: Model computing using physical representations as well as through interactive whiteboards and videos Provide access to video tutorials of computing tasks Select coding apps and websites that allow the students to adjust visual settings (such as font size and contrast) and that are compatible with screen readers	Provide options for *physical action*: Include CS Unplugged activities that show physical relationship of abstract computing concepts Provide teacher's codes as templates Select coding apps and websites that allow coding with keyboard shortcuts in addition to dragging and dropping with a mouse
Build	Provide options for *sustaining effort and persistence*: Teach and encourage peer collaboration by sharing products Utilize pair programming and group work with clearly defined roles Recognize students for demonstrating perseverance and problem-solving in the classroom	Provide options for *language and symbols*: Teach and review content specific vocabulary Teach and review computing vocabulary (e.g. code, animations, computing, algorithm) Post anchor charts and provide reference sheets with images of blocks or with common syntax when using text	Provide options for *expression and communication*: Give opportunities to practice computing skills and content through projects that build on prior lessons Create physical manipulatives of commands, blocks or lines of code Provide options that include starter code
Internalize	Provide options for *self-regulation*: Develop ways for students to self-assess and reflect on own projects and those of others Use assessment rubrics that evaluate both content and process Acknowledge difficulty and frustration. Model different strategies for dealing with frustration appropriately	Provide options for *comprehension*: Provide graphic organizers for students to 'translate' programs into pseudocode Encourage students to ask questions as comprehension checkpoints Use relevant analogies and make cross-curricular connections explicit (e.g. comparing iterative product development to the writing process)	Provide options for *executive functions*: Guide students to set goals for long-term projects Provide exemplars of completed products Provide explicit instruction on skills such as asking for help, providing feedback and using problem-solving techniques

Source: Adapted from Israel et al. (2017).

2018). This guidance is based on a body of research supporting the potential for UDL to increase access and success among students with disabilities in CS education (Israel et al., 2020; Lechelt et al., 2018; Marino et al., 2014; Wille, Century and Pike, 2017). Although the research has thus far been concentrated mostly on UDL implementation in K–12 STEM and postsecondary computer science instruction, scholars have recently identified positive outcomes associated with implementing UDL in K–8 CS instruction (e.g. Hutchison and Evmenova, 2021; Israel et al., 2020).

High leverage practices in CS education

HLPs were designed to support teacher practice and improve student learning outcomes across disciplines for students with disabilities (McLeskey et al., 2019). Although these strategies were developed by the Council for Exceptional Children (CEC) in 2014 with the intent to provide targeted criteria for K–12 special education practitioners, the application of these strategies in any classroom provides tremendous opportunities for all students. The final version of the HLPs included twenty-two instructional approaches separated into four categories (collaboration, assessment, social/emotional/behavioural and instruction) (CEC Division for Early Childhood, 2015; McLeskey et al., 2019). While a growing body of empirical literature on inclusive K–12 CS pedagogies presents findings that point to the use of HLPs as methods for reducing barriers to inclusion, few CS education researchers identified these HLPs explicitly. The following three HLPs were included in this chapter due to their use in the CS education literature.

HLP14: Teaching cognitive and metacognitive strategies to support learning and independence

A quintessential computational thinking strategy is the ability to abstract complex problems and decompose those problems into manageable, smaller tasks (Wing, 2006). Abstraction and decomposition require metacognitive and self-regulatory strategies in which students can activate schema to complete a task. According to Winne (2021), students' use of metacognitive strategies influences how they will handle challenges that emerge during instruction. Therefore, this HLP provides guidance for practitioners to support students' cognitive and metacognitive skill development.

Budin and colleagues (2021) described the reciprocal relationship between the teacher, student and instruction with the intention of students setting achievable goals, monitoring their own progress and adapting to changes throughout instruction. During this process, the teacher models, monitors and selects the strategies to facilitate cognitive development. For example, when employing HLP14 during CS instruction, teachers can assist students systematically and informally with the metacognitive process of debugging. A study conducted by Emara and colleagues (2020) analysed the impacts of the self-regulatory process as well as other problem-solving strategies during targeted debugging activities with students. Students were provided scaffolded activities and encouraged to work together as researchers observed their behaviours during activities. The

regulation and cognitive process results from the study revealed similar attributes to supports offered when employing HLP14, such as providing structure to decompose problems, working collaboratively to evaluate the errors and allowing for open discussion to act on the problem-solving task (Emara et al., 2020).

HLP16: The use of explicit instruction

The use of explicit instruction is defined as the use of a systematic and direct approach to teaching. It is often seen as its own set of instructional practices in special education (Archer and Hughes, 2011). Using this approach to designing CS lessons reduces cognitive load. Examples include breaking down multi-step directions into small manageable chunks and limiting front-loaded teaching into short mini-lessons (Hughes, Riccomini and Morris, 2019: 216). Although this HLP is evident in the K–12 CS education research, like other HLPs, it is not often referenced in those terms. In instances where CS content can seem overwhelming to students, teachers can use explicit instruction to assist in self-monitoring and task completion. Since many CS skills are recursive, designing lessons that review previously taught skills also ensures students retain information and have the opportunity to activate schema to scaffold learning. However, explicit instruction should be balanced with open enquiry approaches so that students have the opportunity to use these skills to engage in more open-ended and creative computational experiences (Israel et al., 2015). Ray and colleagues (2018) found that when teachers provided explicit instruction within enquiry-based activities during CS instruction, such as breaking tasks into simple step-by-step demonstrations and modelling protocols, they increased the engagement of students with disabilities.

HLP19: The use of instructional and assistive technologies

Instructional and assistive technologies can be powerful tools for making instruction accessible, engaging and meaningful. CS teachers should consider using three types of educational technologies when working with students with disabilities: (a) assistive technologies, (b) general instructional technologies and (c) content-specific instructional technologies (Israel and Williams, in press). Although these technologies are often categorized as different types of technologies, there is often an overlap.

While all HLPs are crucial to maximizing student outcomes, in the United States, HLP19 has legal implications as defined by the Individuals with Disabilities Education Act (IDEA; NCES, 2021). Assistive technology is defined in the 2004 reauthorization of IDEA as 'any item, piece of equipment, or product system, whether commercially acquired off the shelf, modified, or customised, that is used to increase, maintain, or improve the functional capabilities of a child with a disability' (IDEA, 2004). These can be high- or low-tech tools, including but not limited to, text-to-speech apps, tablet devices, raised paper with lines and highlighter pens (Israel, 2019). Assistive technology also includes the services required to support the learner in using that technology efficaciously. When considering CS-specific tools, some students will require assistive technologies such as a modified mouse or a text-to-speech app for reading.

Table 10.3 Aligning HLPs to CS education studies

HLP	Example in CS Instruction
#14 Teaching cognitive and metacognitive strategies to support learning and independence	Emara and colleagues (2020) examined how students engaged in debugging from a cognitive and self-regulatory perspective. Progress monitoring, identifying actions and goal setting in collaborative groups facilitated longer stretches of debugging behaviours and increased student outcomes and engagement.
#16 Use of explicit instruction	Taylor (2018) designed a study in which students with intellectual disabilities strengthened computer programming skills when teachers used explicit instructions.
#19 Use of instructional and assistive technologies	Ladner and Stefik (2017) provide a framework in their study to support students with visual impairment through the Quorum programming environment, which is a text-based language that allows for the use of screen readers.

Alongside assistive technologies, instructional technologies that support students with disabilities fall into the following categories: general technologies that are used across content areas (e.g. laptop computers, word processing software) and content-specific technologies (e.g. Scratch). Both types of technology tools can be used to positively support and enhance learning for all students. Thus, as more CS related tools and software become available, ensuring that these technologies are available and accessible to all learners and using them alongside general and assistive technologies that support learning for students with disabilities can enhance student learning outcomes, increase access and ensure legal compliance within the school systems (Table 10.3).

10.3 Next steps: Considering intersectional frameworks

Transforming inclusive CS education relies on developing teachers who understand the intersection of student differences, inclusive pedagogical frameworks and awareness for providing learning opportunities that maximize quality access and engagement for all students. It is the teacher's responsibility to implement pedagogy at the intersection of ability, culture and language within the classroom. In order to support teachers in this endeavour, there must be a unifying framework that attends to these intersections through a multifaceted approach that advances inclusive education beyond merely providing access (Waitoller and Thorius, 2016). Engaging the landscape of inclusive CS education at the intersection of ability, culture/race and translanguaging begins with understanding the interlaced forms of exclusion and recognizing the need for adopting and implementing research-driven frameworks in partnership with researchers, practitioners and policymakers. Inclusive frameworks that guide curriculum and professional development for CS teachers is a pathway for ensuring a greater adoption of classroom practices aimed at closing the equity gaps for marginalized students in CS education (Kapor Center, 2021).

Marginalized intersections of ability and exclusion

The cultural aspect of *disability* has been mis-conceptualized and framed as a deficit among one centralized group of people. Ability, cultural background and language have been historically interlaced and associated with deficiencies and shortcomings throughout the existence of schooling (Artiles, 2011). However, people with disabilities are a diverse mixture of individuals with different identities and a wide range of strengths, experiences and backgrounds who have been forced to use 'normalised' abilities needed to function in school (Waitoller and Thorius, 2016).

Given the diversity of students with disabilities, it is necessary to recognize that students do not experience only one single form of exclusion but rather an overlap of several exclusionary barriers that stand in the way of quality access to, and participation in, learning. Thus, the intersection of students' abilities, cultures and languages make up their unique and complex identities.

Extending strengths of UDL and culturally responsive sustaining education

UDL, culturally responsive sustaining education (CRSE) and translanguaging pedagogy are multifaceted inclusive frameworks that have each contributed to minimizing exclusion barriers for diverse learners in education. We should consider all three equity frameworks because students encounter overlapping layers of barriers that these frameworks, together, can address. A cross-pollination of these inclusionary frameworks, anchored by UDL, can guide practitioners in integrating CRSE and translanguaging in CS instruction, ostensibly extending UDL's benefits beyond providing access and inclusion (Waitoller and Thorius, 2016).

Kapor Center CRSE framework and translanguaging in K–12 CS education

As UDL accounts for access and inclusion for all students, it can be maximized through alignments with the Kapor Center Framework of Culturally Responsive Sustaining Computer Science in providing more quality access for all demographics of learners. This framework for CRSE in CS extends beyond barriers of access by attending to all demographics of culture, language and inequalities positioned within the context of society in CS education (Kapor Centre, 2021). The framework involves explicit implementation of practice within the cultural dimension of learning and the exclusions that have historically enabled students of diverse abilities to be labelled as deficient. Its framework of CRSE in CS extends from culturally responsive teaching (CRT) by addressing barriers in cultural backgrounds of students that range from stereotypes of computer science, learning expectations and the quality of CS classroom inclusion for all students (Kapor Center, 2021). This framework breaks apart the biases of ability through reshaping the constructs of practice for all students to have the opportunity to learn CS. It helps to move teachers towards constructing

inclusive mindsets, by dispelling the historical ones that have marginalized our students' ability, in ways in which the UDL framework can extend its reach of differentiated access to all students.

Translanguaging pedagogy

Working with students from cultural and linguistically diverse backgrounds is a priority which requires situating students in their lived experiences to provide a quality opportunity to participate in deeper learning (Gay, 2000; 2002). Allowing students to include their diverse experiences with language into the classroom enables them to draw meaning from their language as they face new learning experiences, providing a personalized opportunity to expand on their learning. Translanguaging enables students to use their entire range of language to make meaning without being bound to the dominant language, giving students a voice and an opportunity to be included in the learning experience (WIDA, 2020; Wiley and Garcia, 2016). Translanguaging is an inclusive pedagogical approach which provides implications to supporting a joining process of the linguistic ability that students bring into the classroom with the computer concepts they could participate in, which extends UDL's reach beyond access (Vogel et al., 2019).

Cross-pollinating UDL, CRSE and translanguaging in CS education

At the center of equity in inclusive CS education is the belief that all students can succeed academically and that culture, language, ability or socioeconomic status should not present barriers to participation and inclusion (Hansen et al., 2016). Inclusive mindsets are a prerequisite to equitable access which UDL, CRSE and translanguaging offers. To begin considering equity in CS education, teachers must believe that all students should have the opportunity to learn CS and that they can succeed if instruction is designed with them in mind. These equity-focused mindsets lead us to promote the strengths that make up students' identity and to seek equitable frameworks that acknowledge learners' unique intersections (see Figure 10.1).

UDL, CRSE and translanguaging frameworks are intended to guide professional practice, development and preparation for transforming teachers who have adopted the mindsets of inclusive education for closing the equity gaps in computer science. Although professional development (PD) is a pathway to adopt inclusive education practices, majority (80 per cent) PD efforts for inclusive learning address individual forms of student abilities and differences in an isolated approach to exclusion, by assuming that a single form of learner difference is independent from another as opposed to intersecting (Waitoller and Artiles, 2013).

The inclusive frameworks allied together are designed to dismantle the barriers that happen at the unique intersections of ability, culture and language. This alignment can inform pedagogical practices towards providing a wide range of support for diverse learners. However, to achieve a widespread transformation of inclusive CS practices, a streamline of PD for building equitable mindsets for inclusion and developing unified practices of the inclusive frameworks is necessary for nurturing CS teachers to maximize closing the equity gap.

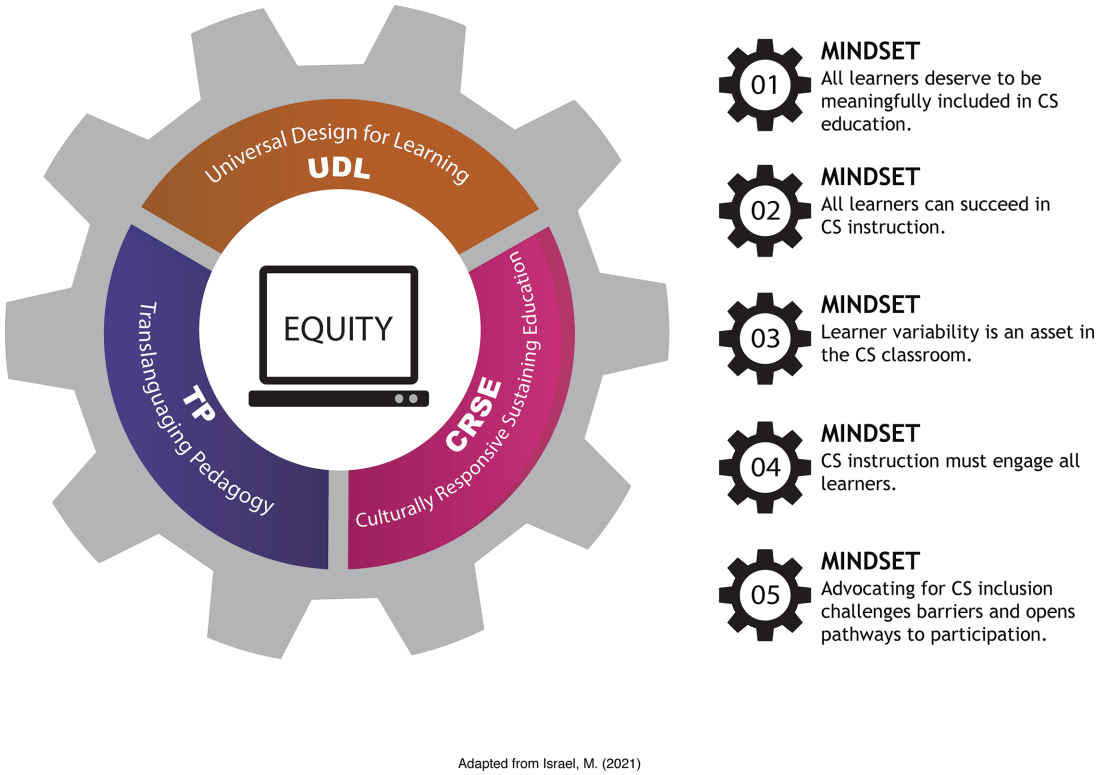

Figure 10.1 Connection of equity frameworks (*Source:* adapted from Israel, 2021)

Key points

- Students with disabilities are a diverse set of learners that cannot be lumped into one category. They have unique strengths and challenges.
- Students with disabilities have often been excluded from K–12 CS education, not because of explicit exclusionary practices, but because of implicitly being left out of CS for All initiatives and opportunities.
- This chapter introduces the idea of an inclusive mindset as well as two ways of proactively planning for the participation and learning of students with disabilities in K–12 CS education: UDL and HLPs. Inclusive mindsets are our beliefs about the belonging of all learners in CS education; UDL and HLPs are instructional approaches that enact full participation and inclusion in CS education.
- In addition to UDL and HLPs, other equity frameworks such as CRSE and translanguaging play a role in inclusive CS education because students are complex and belong in multiple communities.

For further reflection

- What steps can you take to understand both barriers and pathways to inclusion of CS learners in your school system?
- What would it take to move from a deficit-oriented view to a more asset-based perspective of all learners, including those with disabilities in CS education?
- How can you equip teachers with knowledge of Universal Design for Learning (UDL) and high leverage practices (HLP) in CS education to decrease barriers to inclusion in the CS classroom?

References

Archer, A., and Hughes, C. (2011), *Explicit Instruction: Effective and Efficient Teaching,* New York: Guilford.

Artiles, A. J. (2011), 'Toward an Interdisciplinary Understanding of Educational Inequity and Difference: The Case of the Racialization of Ability', *Educational Researcher*, 40 (9): 431–45. https://doi.org/10.3102%2F0013189X11429391.

Basham, J. D., Israel, M., Graden, J., Poth, R., and Winston, M. (2010), 'A Comprehensive Approach to RTI: Embedding Universal Design for Learning and Technology', *Learning Disability Quarterly*, 33 (4): 243–55. https://doi.org/10.1177/073194871003300403.

Bronfenbrenner, U. (1977), 'Toward an Experimental Ecology of Human Development', *American Psychologist*, 32 (7): 513.

Budin, S., Hashey, A., Patti, A., and Rafferty, L. (2021), 'Translating High-leverage Practices to Remote Environments: Tips for Teacher Educators', *Journal of Special Education Preparation*, 1 (1), 25–35.

Burgstahler, S. (2020), *Equal Access: Universal Design of Instruction*, University of Washington: DO-IT. https://www.washington.edu/doit/sites/default/files/atoms/files/EA_Instruction_5_28_20_0.pdf.

CAST (n.d.), UDL in Public Policy. https://www.cast.org/impact/udl-public-policy.

CAST (2018), UDL and the Learning Brain. https://www.cast.org/binaries/content/assets/common/publications/articles/cast-udlandthebrain-20220228-a11y.pdf.

CEC Division for Early Childhood (2015), *DEC Recommended Practices: Enhancing Services for Young Children with Disabilities and Their Families*, Arlington, VA: Author.

Emara, M., Grover, S., Hutchins, N., Biswas, G., and Snyder, C. (2020), Examining Students' Debugging and Regulation Processes during Collaborative Computational Modeling in Science. *ICLS*.

Every Student Succeeds Act, no. 20 U.S.C. § 6301 (2015), https://www.congress.gov/114/plaws/publ95/PLAW-114publ95.pdf.

Florian, L., and Black-Hawkins, K. (2011), 'Exploring Inclusive Pedagogy', *British Educational Research Journal*, 37 (5): 813–28.

Gay, G. (2000), *Culturally Responsive Teaching: Theory, Research, and Practice*, New York: Teachers College Press.

Gay, G. (2002), 'Preparing for Culturally Responsive Teaching', *Journal of Teacher Education*, 53 (2): 106–16. https://doi.org/10.1177%2F0022487102053002003.

Grover, S., and Pea, R. (2013), 'Computational Thinking in K–12: A Review of the State of the Field', *Educational Researcher*, 42 (1), 38–43. https://doi.org/10.3102/0013189X12463051.

Hansen, A. K., Hansen, E. R., Dwyer, H. A., Harlow, D. B., and Franklin, D. (2016), 'Differentiating for Diversity: Using Universal Design for Learning in Elementary Computer Science Education', in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, 376–81: ACM. https://doi.org/10.1145/2839509.2844570.

Hughes, C. A., Riccomini, P. J., and Morris, J. R. (2019), 'Use Explicit Instruction', in J. McLeskey (ed.), *High-Leverage Practices in Special Education,* 215–36, Arlington, VA: Council for Exceptional Children and CEEDAR Center.

Hutchison, A., and Evmenova, A. S. (2021), 'Planning Computer Science Instruction for Students with High-Incidence Disabilities', *Intervention in School and Clinic*, 57 (4), 262–7. doi: 10534512211024939.

Individuals with Disabilities Education Act, no. 20 U.S.C. § 1400 (2004), http://uscode.house.gov/view.xhtml?path=/prelim@title20/chapter33&edition=prelim.

Irwin, V., Zhang, J., Wang, X., Hein, S., Wang, K., Roberts, A., York, C., Barmer, A., Bullock Mann, F., Dilig, R., and Parker, S. (2021), *Report on the Condition of Education 2021* (NCES 2021-144). U.S. Department of Education. Washington, DC: National Center for Education Statistics. https://nces.ed.gov/pubsearch/pubsinfo.asp?pubid=2021144.

Israel, M. (2019), 'Use Assistive and Instructional Technologies', in J. McLeskey, L. Maheady, B. Billingsley, M. T. Brownell, and T. J. Lewis (eds), *High Leverage Practices for Inclusive Classrooms,* 264–278, New York: Routledge.

Israel, M. (2021), 'Equity Principles for Including Learners with Disabilities in K-12 CS Education', in *Understanding Computing Education (Vol 2): Equity, Diversity and Inclusion. Proceedings of the Raspberry Pi Foundation Research Seminars.*

Israel, M., Jeong, G., Ray, M., and Lash, T. (2020), 'Teaching Elementary Computer Science through Universal Design for Learning', in *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, 1220–6. https://doi.org/10.1145/3328778.3366823.

Israel, M., Kester, M., Williams, J., and Ray, M. (in press), 'K-12 Teachers' Confidence in Supporting Students with Disabilities in Computer Science Education: The Influence of Instructional Coaches', *Transactions on Computing Education.*

Israel, M., and Ladner, R. E. (2016, September), ' "For All" in "Computer Science For All" ', *Communications of the ACM*, 59 (9): 26–8.

Israel, M., Lash, T., and Ray, M. (2017), 'Universal Design for Learning within Computer Science Education'. Creative Technology Resource Lab, University of Florida. https://ctrl.education.ufl.edu/wp-content/uploads/sites/5/2020/05/Copy-of-UDL-and-CS_CT-remix.pdf.

Israel, M., Wherfel, Q. M., Pearson, J., Shehab, S., and Tapia, T. (2015), 'Empowering K–12 Students with Disabilities to Learn Computational Thinking and Computer Programming', *TEACHING Exceptional Children*, 48 (1): 45–53.

Kapor Center (2021), Culturally Responsive-Sustaining Computer Science Education: A Framework. https://www.kaporcenter.org/equitableCS/.

Kirby, M. (2017), 'Implicit Assumptions in Special Education Policy: Promoting Full Inclusion for Students with Learning Disabilities', *Child & Youth Care Forum*, 46, 175–91. https://doi.org/10.1007/s10566-016-9382-x.

Ladner, R., and Stefik, A. (2017), 'AccessCSforAll: Making Computer Science Accessible to K–12 Students in the United States', *SIGACCESS Newsletter*, 118, 3–8.

Lechelt, Z., Rogers, Y., Yuill, N., Nagl, L., Ragone, G., and Marquardt, N. (2018), 'Inclusive Computing in Special Needs Classrooms: Designing for All', in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 1–12. https://doi.org/10.1145/3173574.3174091.

Marino, M. T., Gotch, C. M., Israel, M., Vasquez, E., Basham, J. D., and Becht, K. (2014), 'UDL in the Middle School Science Classroom: Can Video Games and Alternative Text Heighten Engagement and Learning for Students with Learning Disabilities?' *Learning Disability Quarterly*, 37 (2): 87–99.

McLeskey, J., Billingsley, B., Brownell, M. T., Maheady, L., and Lewis, T. J. (2019), 'What Are High-Leverage Practices for Special Education Teachers and Why Are They Important?' *Remedial and Special Education*, 40 (6), 331–7. https://doi.org/10.1177/0741932518773477.

Office of Educational Technology (2017), 'Reimagining the Role of Technology in Education: 2017 National Education Technology Plan Update' (policy brief), U.S. Department of Education. https://tech.ed.gov/files/2017/01/NETP17.pdf.

Posey, A. (n.d.), *Universal Design for Learning (UDL): A Teacher's Guide*, Understood for All Inc. https://www.understood.org/articles/en/understanding-universal-design-for-learning.

Ray, M. J., Israel, M., Lee, C., and Do, V. (2018), 'A Cross-Case Analysis of Instructional Strategies to Support Participation of K-8 Students with Disabilities in CS for All', in *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 900–5. https://doi.org/10.1145/3159450.3159482.

Rose, D. H., Meyer, A., and Hitchcock, C. (2005), The Universally Designed Classroom: Accessible Curriculum and Digital Technologies, Cambridge, MA: Harvard Education Press.

Snodgrass, M. R., Israel, M., and Reese, G. C. (2016), 'Instructional Supports for Students with Disabilities in K-5 Computing: Findings from a Cross-Case Analysis', *Computers & Education*, 100: 1–17.

Taylor, M. S. (2018), 'Computer Programming with Pre-K through First-Grade Students with Intellectual Disabilities', *Journal of Special Education*, 52 (2): 78–88.

Tissenbaum, M., Sheldon, J., and Abelson, H. (2019), 'From Computational Thinking to Computational Action', *Communications of the ACM*, 62 (3), 34–6.

Vogel, S., Hoadley, C., Ascenzi-Moreno, L., and Menken, K. (2019), 'The Role of Translanguaging in Computational Literacies: Documenting Middle School Bilinguals' Practices in Computer Science Integrated Units', SIGCSE 2019', in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 1164–70. doi: 10.1145/3287324.3287368.

Waitoller, F. R., and Artiles, A. J. (2013), 'A Decade of Professional Development Research for Inclusive Education: A Critical Review and Notes for a Research Program', *Review of Educational Research*, 83 (3): 319–56. https://doi.org/10.3102/0034654313483905.

Waitoller, F. R., and King Thorius, K. A. (2016), 'Cross-Pollinating Culturally Sustaining Pedagogy and Universal Design for Learning: Toward an Inclusive Pedagogy That Accounts for Dis/ability', *Harvard Educational Review*, 86 (3): 366–89.

WIDA (2020, September), 'Translanguaging: Teaching at the Intersection of Language and Social Justice'. www.wida.wisc.edu. https://wida.wisc.edu/sites/default/files/resource/Focus-Bulletin-Translanguaging.pdf.

Wiley, T. G., and Garcia, O. (2016), 'Language Policy and Planning in Language Education: Legacies, Consequences, and Possibilities', *Modern Language Journal*, 100 (1): 48–63.

Wille, S., Century, J., and Pike, M. (2017), 'Exploratory Research to Expand Opportunities in Computer Science for Students with Learning Differences', *Computing in Science Engineering*, 19 (3): 40–50. https://doi.org/10.1109/MCSE.2017.43.

Wing, J. M. (2006), 'Computational Thinking', *Communications of the ACM*, 49 (3): 33–5.
Winne, P. (2021), *Cognition, Metacognition, and Self-Regulated Learning*. Oxford Research Encyclopedia of Education. Oxford: Oxford University Press. doi:10.1093/acrefore/9780190264093.013.1528
Zirkel, P. (2019), 'A Blanket Approach to 504 Plans Does a Disservice to Students and the School', *National Association of Secondary School Principals*, vol. 19. https://www.nassp.org/publication/principal-leadership/volume-19-2018-2019/principal-leadership-march-2019/testing-accommodations-for-students-with-disabilities/.

Part 3

Teaching and Learning in Computer Science

11

Introduction to Part 3

Erik Barendsen

There are two main angles for approaching teaching and learning in computer science. First, one can start from general educational principles and theories and explore how they appear in (and apply to) computer science education. In contrast, one can focus on specific computing content matter and investigate aspects of topic-specific pedagogies relative to this content, such as possible learning goals and objectives, factors influencing learners' understanding, instructional strategies, and assessment. This part takes the first angle, whereas Part 4 follows the second approach by focusing on pedagogical aspects of programming.

As computer science is being introduced on more and more educational levels, understanding learners' possibilities and challenges and developing age-appropriate pedagogies are becoming crucial. In Chapter 12, Tim Bell and Caitlin Duncan discuss principles for teaching computing content at primary school level, advocating to integrate computer science content into more traditional content matter.

In Chapter 13, Paul Curzon and colleagues explore strategies to foster students' conceptual learning. They present a variety of possible approaches, ranging from using analogies to more elaborate classroom practices such as enquiry-based learning. The semantic waves theory is used as a learning-theoretical principle to explain the effectiveness of some of the presented learning activities.

Chapter 14 moves on to the role of language in teaching and learning. One way of looking at learning a topic is to regard it as learning a new – professional – language. Ira Diethelm and co-authors explore students' everyday language, the computer science scientific language and ways to scaffold the transition between the two. A model of discourse ('talk') in the computing classroom plays a central role in the chapter.

Understanding learners' attitudes towards the discipline and towards learning is crucial for designing appropriate learning environments. In Chapter 15, Quintin Cutts and Peter Donaldson focus on the interesting notion of mindset as a factor influencing learners' success. Classroom practices and, surprisingly, teachers' own mindsets turn out to be instrumental to the development of the learners' mindsets. The authors confront the reader with key aspects of their own mindset and discuss results of scientific research on mindsets of computer science students and teachers.

Chapter 16 focuses on assessment in the computing classroom. In accordance with the recent shift in attention from summative to formative assessment, Sue Sentance, Shuchi Grover and Maria Kallia present possibilities to implement the latter, also known as 'assessment for learning'. The reader learns about general principles such as peer assessment as well as concrete tools such as concept mapping. Moreover, pointing ahead to Part 4, this chapter presents specific approaches towards formative assessment of programming projects.

12

Teaching Computing in Primary Schools

Tim Bell and Caitlin Duncan

Chapter outline

Chapter synopsis

With the introduction of computing into primary school curricula in many countries, and as informal learning opportunities in others, specific pedagogies to support teaching this topic to primary-aged children need to be examined. In this chapter we will summarize current research in this area and discuss our own experiences in New Zealand, working with primary school teachers and students.

12.1 Introduction

Classroom pedagogy for teaching computing is something that changes considerably from early school years through to the end of high school. Extending computing to the primary years of school (students of ages five through to twelve to thirteen years old) presents unique challenges as well as opportunities. In this chapter we will look at what computing would look like in a primary school and reasons why it should be taught at that level. We focus on elements relating to computational thinking and concepts from computer science.

A common concern around incorporating this subject into primary school is that the concepts it covers are too advanced and are unsuitable for this age group. There is also the issue that teachers are already working with a 'crowded curriculum', so this subject could just become another add-on that there isn't time to teach. The vast majority of teachers do not have prior knowledge of this subject, so along with finding time to teach it, they need time for their own learning as well.

To avoid this new subject at primary school coming across as a seemingly unrelated collection of topics, it is useful to be aware of the *big ideas* that we want students to take away from their learning (Bell, 2016). For example, many primary school students learn a programming language such as Scratch, but what are the concepts that we want them to learn from this? In ten years' time it is unlikely that they will need to know the name of the command for storing a value into a Scratch variable. Likewise, many curricula include converting binary numbers to decimal, but in practice, few people may ever have to do that. Yet there is value in teaching both Scratch programming and binary number representation. Looking for the big picture will be a theme of this chapter, since we need to appreciate the overarching goals to be able to make sense of the small details that appear in the classroom.

Computational thinking (CT) has been described in previous chapters (e.g. Chapter 5), and through this chapter we will connect this definition with primary school curriculum. Before defining and exploring these ideas carefully, we begin with a case study to give us something concrete to illustrate the principles and ideas that we will introduce.

12.2 Case study one: Binary representation

Data representation and 'binary numbers' are topics that merit understanding by students, as binary representation is fundamental to how *everything* is stored, manipulated and communicated on digital devices. In fact, one of the defining characteristics of digital technology is in its name: everything is represented by digits!

Jargon such as 'binary numbers' may put teachers off approaching the subject of computer science. One of the keys to making computer science (CS) and CT work in primary school is making it explicit that much of this jargon can be seen simply as 'big words for simple ideas'; every specialist field needs technical language to avoid having to use long-winded descriptions of commonly used ideas.

Example: Teaching binary representation to early primary children using CS Unplugged

This activity is described more fully in an open-source lesson (CS Unplugged, n.d.).

- Create a set of five cards, with dots on one side and nothing on the other. The cards have 1, 2, 4, 8 and 16 dots, respectively.
- Place them down in order with the one-dot card on the far right and the one with sixteen dots on the far left.
- Give the rule that each card either has all of its dots visible or is flipped to show none.
- Using this context, students can be scaffolded to discover binary representations (e.g. 'show exactly five dots') and to explore limits (such as thirty-one being the maximum number that five cards can show).

The binary representation activity shows how the concept of representing numbers and letters in binary can be communicated to relatively young students by manipulating cards (Figure 12.1). It is taught with students constructing the concepts themselves, rather than simply being told how things work. For example, they can work out for themselves that each card has twice as many dots as the previous one. In doing this, students are also demonstrating the CT skills of logical reasoning and pattern recognition.

Initially 'yes' and 'no' are used to communicate if each card has its dots visible or not, and the terms 'zero' and 'one' for the two symbols aren't mentioned until later, which emphasizes that they are an arbitrary abstraction for what is physically happening on a computer with electrical charges and magnetism. The important thing isn't what the two symbols are; it is the fact that using *any* two different symbols allows us to represent any type of data (e.g. the activity quickly extends to

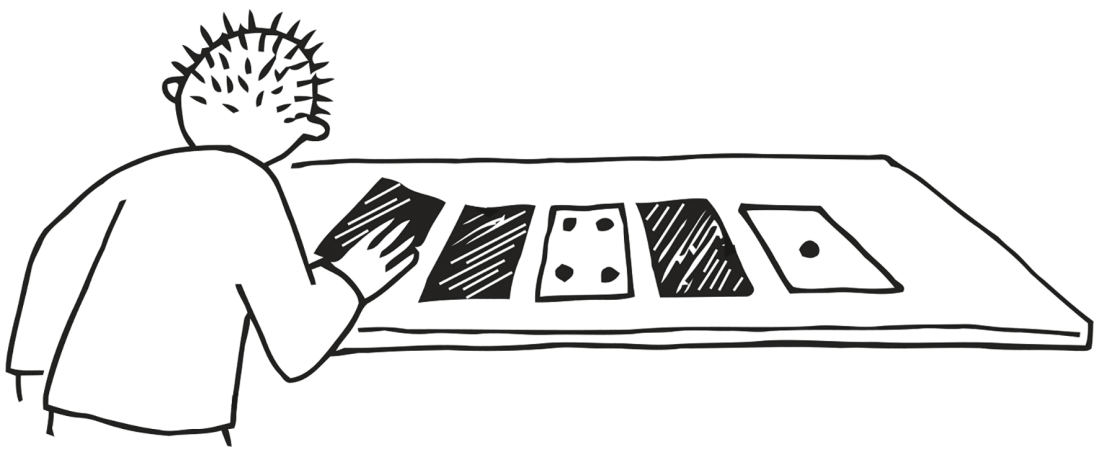

Figure 12.1 Interacting with binary numbers using cards

representing letters of the alphabet using numbers). The technical word for these two values is 'binary digit', abbreviated as 'bit', and while it is good for students to know this terminology, most of their time has been spent understanding the implications of binary representation.

Sometimes 'binary numbers' are taught as simply converting conventional decimal numbers to a representation with zeroes and ones, which can be conveniently assessed in exams. In practice, computer scientists rarely convert numbers between binary and decimal; the real concepts at play here are much deeper. The following section discusses one of the key ideas that students can take away from the activity above.

Anything stored on a computer can be represented using just two symbols

This is a powerful concept and is easily demonstrated to students using the numbers to represent months. Without explanation, the teacher says something like, 'The month I was born in is no-no-yes-yes-yes' (pointing at the cards from left to right), which students could translate to 00111, or the decimal number 7, but are likely to recognize it as 'July'. This gives the opportunity to point out that a new type of data (months in the range January to December) can be represented by simply saying yes and no. From a CT point of view, this is an application of abstraction; the name 'July' carries a lot of extra information and history – for example it was named after Julius Caesar – but for practical purposes (such as working out how long it is until my birthday), abstracting the word to number 7, or binary representation 00111, is sufficient.

Another important concept is the range of values that can be represented by a given number of bits. The above activity uses five cards and can represent any value from zero to thirty-one. This gives thirty-two *different* values (a concept that may be a challenge for some students to come to grips with). But with some guidance, students should be able to extend this idea to six cards, which represent the values from zero to sixty-three, giving sixty-four different values. The principle that this has started to expose is discussed in the following section.

Each extra bit added to a binary representation doubles the range of values that can be represented

This observation comes up in many different areas. For example, 5 bits provides thirty-two different values and so are sufficient to represent the twenty-six letters of the English alphabet (with some combinations left over). However, this isn't enough to represent the approximately 100 characters used on an English computer keyboard, which includes upper and lower case, punctuation and other symbols. For this 7 bits (allowing 128 different values) are needed.

Extending this to languages such as Chinese, which have tens of thousands of symbols, one might expect that a very large number of bits is needed for each character. In fact, just 16 bits for each character allows for 65,536 different symbols and is sufficient for most documents. The same

also applies to representing colours. It is generally accepted that the human eye can distinguish a few million different colours at most. A 24-bit representation of colour allows a digital device to store over 16 million different colours, and so in principle it is more accurate than the human eye can perceive. Using more than 24 bits would be redundant.

Relating the technical idea of binary numbers to helping humans communicate in their own language, or perceive the colours in images accurately, gives meaning to what otherwise appears as a purely technical concept. These are highly motivating examples for both students and teachers. This isn't just an interesting example though; it is the whole point of binary representation: to be able to store things as diverse as words, images, sounds and financial information at a level of accuracy that matches human needs.

Key concept: Digital systems should be designed for humans

The previous example makes the point that binary numbers relate directly to human needs and values. This idea applies to most technical areas of computing: interfaces need to be designed to interact well with the way that the user thinks and works, and programs need to be written in a way that if another human has to read the 'code', they can do so easily. Computer systems need to be designed to be reliable so that people aren't constantly wasting their time having to get material from backups or asking for a repeated download. They also must be reliable in terms of safety, as digital systems are used in high-risk environments such as medical care. Networks need to operate at a human timescale – we are used to communicating and thinking at the level of around 1 second; if a system constantly takes 30 seconds to respond, it will be frustrating and difficult to use. We summarize this with the mantra: computer programs aren't written for computers; they are written for humans.

12.3 Computational thinking, computer science and programming at the primary school level

A computing curriculum should support the development of CT skills. As discussed in Chapter 5, CT has been defined in many different ways, although the varying definitions are broadly in agreement.

It's important that teachers are clear on what CT is and isn't, since not everything that happens on computers will be CT and vice versa. The value of being able to work with computational ideas is applicable beyond being able to program and, conversely, can be learnt in areas other than programming (such as the 'Unplugged' example above). However, converting an idea to a program and getting it to work on an autonomous device fully exercises one's ability to be precise in the expression of the steps needed to solve a problem (Hermans and Aivaloglou, 2017). Programming is an excellent way of consolidating and testing one's CT skills.

Having acted out ideas in an unplugged context enables students to properly understand the steps of their algorithm. This can then map directly onto an implementation, since Unplugged activities enforce the constraints that computation imposes on a programmer (Bell, 2021). It also helps with debugging, since this can be done effectively only if the programmer understands what the program was intended to do at each step. It is important to note that teaching *only* in an unplugged style has disadvantages, just as teaching *only* programming does. When taught in isolation, students (and sometimes teachers) may perceive unplugged activities as isolated learning activities or even just games, and they can struggle to make connections between the activities and computing (Duncan, 2019). Preceding learning programming with Unplugged activities has been demonstrated to increase students' self-efficacy (Hermans and Aivaloglou, 2017). Programming contextualizes unplugged and vice versa (Bell, 2021). A survey of research on unplugged approaches can be found in Bell and Vahrenhold (2018).

In the following we reflect on how an activity, such as the binary number activity, can exercise common aspects of CT for students at a primary school level, based on the outline given in Chapter 5. A more detailed description of these connections can be found at the end of the lesson plans on csunplugged.org.

Logical thinking

Logical reasoning is about trying to make sense of things by observing, thinking about the facts and rules that you know are correct and using logic to deduce more rules and information from these. In our binary numbers example, students used these skills to evaluate how many numbers can be represented with a certain number of bits or how to represent a given number.

Algorithmic thinking

This supports students to follow algorithms and to create algorithms to solve problems by breaking them down into steps. In our binary example, students practise and develop algorithmic thinking skills as they learn (by constructing the knowledge themselves) an algorithm for converting decimal numbers to binary and practise following this algorithm. When looking for evidence of students' algorithmic thinking skills, teachers can observe how methodical students are in their approach to the task.

Pattern recognition

Finding patterns is an important part of working in computation because repeated patterns can be automated and generalized. For the binary number activity, there are obvious patterns in the number of dots on each card as well as the maximum value that can be represented by a given number of cards. There are multiple other patterns that can be explored within this activity as well.

Abstraction and generalization

This is about simplifying things by identifying what is and is not important and removing all specific details and patterns that will not help us solve our problem. By doing this we can create problem representations and solutions that are as general as possible. The binary activity requires students to work with several abstractions, including using numbers to represent letters of the alphabet, cards to represent the digits 0 and 1, and even the terms 'zero' and 'one', or 'off' and 'on', to represent the two different states that the digits can be in. Binary number representation itself is an abstraction! It hides the complexity of the electronics and hardware inside a computer that store data, and the versatility of digital devices hinges on the generalization possible through this simple abstraction.

Evaluation

This is about identifying the possible solutions to a problem and judging which is the best to use. For the binary numbers, students can evaluate how many different values can be represented with 5 bits, then with 6 bits and more. The result of this evaluation is that they should see the exponential growth in the range of values as the number of bits increases and the trade-off between richness of representation with the cost of storage space.

12.4 Reasons for introducing computing in primary schools

There are several positive consequences of having students experience computing at primary school age, as opposed to encountering it for the first time as an adolescent or later (Duncan, Bell and Tanimoto, 2014):

- Students are exposed to the concepts and ideas when they are younger, before misconceptions about the nature of the subject set in (e.g. 'It's just for boys', 'It's about sitting in front of a computer and using it all day'); these misconceptions potentially damage students' views of their competence in computing, particularly those from historically underrepresented or excluded groups.
- Diversity is increased by giving as many students as possible exposure and experience in computing, not just those with a special interest, or extra opportunities and privilege.
- It provides a chance for students to find out which skills are closely connected with programming and computing in general, particularly maths and communication skills, which are harder to pick up if a student develops a passion for computing later in their schooling or career.
- Natural languages are learned most easily before approximately ten years of age, and while the same has not yet been fully established for programming languages, there is some evidence that this may be the case.

- CT and programming provide good opportunities for an integrated curriculum, where students can learn subjects as diverse as maths, literacy and music in ways that can be deeper than learning the subjects in isolation (Duncan, Bell and Atlas 2017). An integrated curriculum is generally easier to implement in junior years because it is common for students to have one teacher for all subjects, rather than multiple teachers covering single subjects, as is common in intermediate and high school.

12.5 The bigger picture

Having looked at some specific examples and considered the role of CT in the curricula, we now take a step back and think again about why particular topics should appear in curricula for beginning students. We begin by thinking about the ecosystem surrounding digital devices, so that we have a benchmark to compare curriculum topics against.

Figure 12.2 shows a model of a digital system based on six key elements that are commonly found in curricula (Duncan and Bell, 2015). The 'digital device' would traditionally be a computer, but it could also be a smartphone, a tablet or an embedded device such as a burglar alarm or cash register. All of these devices run *applications*, otherwise referred to as software, apps or firmware; examples would be a word processor, clock, web browser or weather simulator as well as include hidden software such as a printer driver or a washing machine controller. These applications apply an *algorithm* to *data*; the algorithm might be as simple as incrementing the number of steps taken in a personal fitness device or as complex as predicting the weather 1 week from now. A key is that

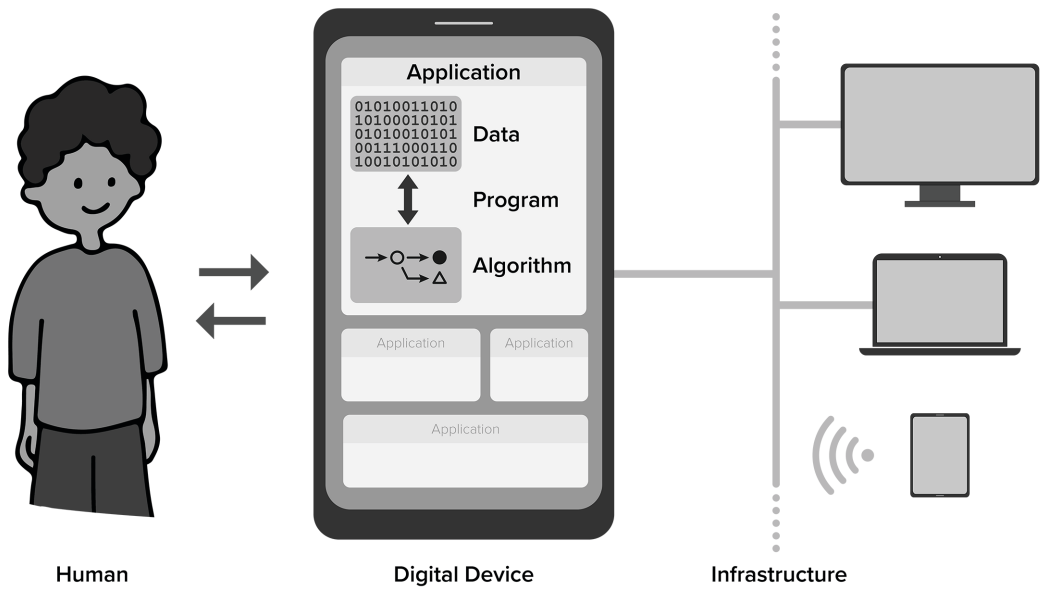

Figure 12.2 A model of digital systems

the *algorithm* is just the process that happens, but to implement the algorithm, someone needs to create a *program* that physically operates on the data, in a form that the digital device can execute. Many students may not even be aware that all these devices are running computer programs; the tools in common use, including search engines and social networks, can appear as monolithic entities rather than 'code' that someone has actually written.

The most important component of the system is the human, as software and hardware are created to address a human need. This need might be as trivial as entertainment – playing a small game or an animated movie – through making sure that food can be delivered to a population economically and quickly. In all these cases, the interface to the human is crucial; a confusing interface can lead to disasters, and a delightful interface enables the software and its associated hardware to be sold at a premium.

Finally, very few digital devices exist in isolation, and they are connected through some sort of *infrastructure*, whether it is the internet, a local area network, a small personal network such as Bluetooth or a 'sneakernet', where a memory card or disk is physically moved between devices such as from a camera to a computer.

The six components in Figure 12.2 provide a model for the knowledge that students will need in order to fully understand a system and be able to design their own digital artefacts. A complaint about older curricula is that they focus primarily on the applications, while the data, algorithms, programs and infrastructure are treated as a black box, and the human is expected to conform to the system, rather than viewing the interface critically and considering what is good about it and what might be improved.

This provides us with a bigger picture: if we can explicitly expose students to all six elements of digital systems in a form that is meaningful in their world, then we can give them a better understanding of how it all works and empower them to be technology *creators* rather than just *consumers*. This now puts a topic such as 'binary numbers' in context; for example, students can gain an appreciation that the same hardware can be used for working with a wide variety of information and that it is binary data moving around the system that makes great things happen. It also explains why inventions like flash memory open new possibilities for photographers, musicians and seismologists alike because the same physical memory can be used for such different purposes.

Moving back to the bigger picture of a device, and the view that the human is the most important part of a computer system, we become aware that as students learn about programming, they need to always have the human (i.e. the user) in mind. This can be as trivial as thinking about how a response will affect the user, such as a rather humiliating 'You are wrong!!!' response to an incorrect answer to a quiz question or making users wait unnecessarily because the programmer couldn't be bothered using a more efficient algorithm.

12.6 Integrated learning

Introducing computing to primary education is, for many teachers and schools, an intimidating task. One approach that can be helpful is to use integrated learning, where computing concepts are learned in the context of other subjects and vice versa.

Integrated learning can appear in many forms in computing classes. For example, investigating early cryptographic methods can open an opportunity for student inquiry that is relevant to social studies around the use of codes in wartime. Writing programs to play music can utilize skills from both subjects at the same time (Bell and Bell, 2018). Lee, Martin and Apone (2014) integrate computing lessons with storytelling, science investigations and analysing locational data, while Smith and Burrow (2016) use computing in the context of story writing.

There are several benefits of using integrated learning. One is that lessons use partially familiar ideas; for example, teaching programming in the context of music for a teacher who is confident with music means that just the programming is the new knowledge domain for the teacher; in contrast, having students write a program to convert binary numbers could have both the teacher and students working with two new knowledge domains.

Another advantage of integrated learning is that the new subject of computing can have a positive impact on the existing curriculum, rather than competing for valuable time. For example, teaching geometrical ideas like distance and angles through a language like Scratch can be motivating for students because the system helps them visualize and quickly apply what they are learning, and at the same time, they are learning to sequence and possibly iterate commands (Duncan and Bell, 2015, Duncan 2019).

12.7 Case study two: Programming – what are the main concepts?

A lot of the improvements in curricula focus on learning programming (or 'coding'), so we provide a second case study here to unpack the concepts at play at the primary school level.

Example: Kidbots

In the 'Kidbot' activity (https://www.csunplugged.org/en/topics/kidbots/unit-plan/), a large grid (typically an 8 by 8 grid of 30 centimetre squares; see Figure 12.3) is used on which students can step out simple instructions like 'Forward', 'Turn left' and 'Turn right'. Three students play the roles of 'Bot', 'Programmer' and 'Tester', respectively. The 'Bot' stands on a square in the grid, and another square is identified that they need to get to. The programmer writes a series of instructions using the commands and then gives their 'program' to the tester to read them to the bot. Once the tester gets the program, it is executed without any adjustment, to simulate what happens when a program is run on a computer (this is why a tester is needed; the programmer will be very tempted to make changes on the fly!). The course can be made progressively more difficult by adding obstacles, and extra challenges include finding alternative sets of instructions that achieve the same thing.

A follow-up activity is to do the same thing with a 'turtle'-based system, either one with physical movement such as a Bee-Bot or an on-screen one such as the Bee-Bot app.

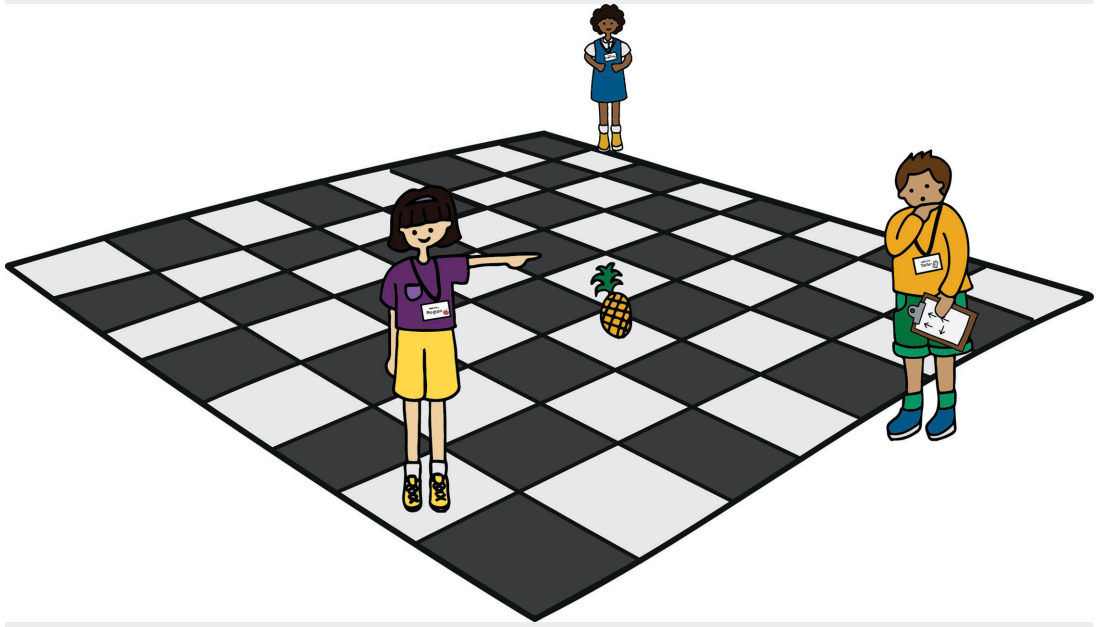

Figure 12.3 Kidbot grid

The 'Kidbot' activity enables students to physically experience what they can later program a device to do. Although the two activities are the same at an abstract level, doing the activity away from the computer encourages the student to think through their program. This helps prevent them from entering into a kind of 'programming by permutation' process where they continually make small changes to a program without understanding it, in the hope that it will fix a mystery bug that they have encountered.

Introductory programming systems such as the turtle-based approach described here, as used in languages like Scratch, Snap!, Blockly, Logo and so on, work with concrete ideas based around movement that students will be familiar with, yet can introduce fundamental ideas like sequence, selection (if statements) and iteration (loops). This is a good example of scaffolding using current knowledge (movement in a 2D space) to teach some simple programming commands.

A valuable aspect of starting with unplugged exercises is that it is an opportunity to use technical terminology appropriately: the device follows an *algorithm* that is implemented by *programming* the device, and if *testing* shows that it doesn't work correctly, then the *program* needs to be *debugged*. The importance of learning the language of the discipline is discussed further in Chapter 14.

In the bigger picture of learning programming, the simple 'Kidbot' programming language introduces several important programming concepts: the use of sequence as well as testing and debugging. However, it is important for students to advance (eventually) beyond the concept of *sequence* to become comfortable with *selection* and *iteration*. Together, these three concepts

give access to the full power of computing (Aho, 2010), and if we consider a typical beginner's programming language like Scratch, Snap! or Logo, we can see that it contains all the elements and, therefore, is suitable for teaching programming in considerable depth, *provided* that all those elements are exercised well. It appears that despite the potential of such languages, they aren't widely used to explore the full power of programming. For example, Aivaloglou and Hermans (2016) found that in a sample of around 4 million Scratch programs shared online, 78.33 per cent had *no* decision points and 13.8 per cent had only one, so very few student programs were using 'if' statements or conditional loops and therefore weren't accessing the full power of a computational device. Likewise, Amanullah and Bell (2019) found that only about a third of Scratch programs used decision points, and the use of common programming patterns was very rare.

Being aware of the key elements of computer programs (and computing in general) dispels the myth that computing is changing so fast that knowledge will be out of date too quickly; on the contrary, the fundamental elements of modern programming languages that are used to program the latest devices would be recognized by Alan Turing and are no more powerful computationally than the nature of computation that he articulated in the 1930s (although they are smaller and faster!)

Thinking about programming as teaching these key elements means that teachers should see themselves as teaching *programming*, rather than teaching a particular language. While animations or games that don't use all of these elements in any depth can be inspiring for students and have cross-curricular value, a teacher shouldn't lose sight of enabling students to work with the elements at an age-appropriate level. For a very young student, this might involve focussing only on sequence and possibly iteration (e.g. repeat commands), in the same way that in maths the preparation for algebra is to learn basic facts about numbers, in preparation for learning more powerful notation later.

More details around teaching 'coding' are provided in Part 4, but it is useful to note that in primary schools the role and context of programming as a subject is changing as it becomes a part of mainstream curricula. There is a view that younger students can gain a lot through playful interaction with a programming environment; Resnick and Rusk (2020) point out that 'although it might seem more efficient to teach concepts through direct instruction, we have seen that many students become more engaged and gain a greater sense of agency and confidence when they learn through playful experimentation and exploration'. However, the advent of curricula in many countries has led to more formal expectations, and an important trend is the understanding that *reading* and interacting with existing code is a valuable prelude to *writing* it. This is reflected in approaches such as 'Predict, Run, Investigate, Modify, Make' (PRIMM) (Sentance, Waite and Kallia, 2019).

Through a focus on CT rather than on just programming, students will be learning generally applicable skills (such as giving clear and unambiguous instructions) as well as realizing that computers do only what a program says and not necessarily what the programmer intends. Another key idea is that most programs will need debugging, and the act of programming has to include this. It is appropriate at the primary school level to be getting familiar with these ideas as general skills and to appreciate what a program is. The goal of this is not to have every student enter a career in programming, but to ensure students have the opportunity to understand the world they are living in, where digital systems are a part of nearly everything they do.

12.8 Teacher education

Providing large-scale and sustainable professional development, resources and support to teachers is yet another challenge. However, all of these can be overcome with suitable support (Brown et al., 2014; Falkner, Vivian and Falkner, 2014; Schulte, Hornung and Sentance, 2012; Bell, Duncan and Atlas, 2016; Duncan, 2019). For primary school teachers, professional development needs to be in the context of helping a generalist teacher see the value of adding another topic to the range they need to cover; beyond primary school, it can involve developing more advanced knowledge (including programming) and particularly the pedagogy to deliver these topics.

Pre-service education can be a major challenge; colleges of education are often contending with an already full curriculum, so fitting in a new subject that the academic staff aren't experienced teaching can be just as challenging as introducing it in schools. This adds to the need for training and resources that can be implemented quickly by teachers.

Support for teachers new to the area can be provided in many ways: in-person workshops, online courses including massive open online courses (MOOCs), peer mentoring and co-teaching, subject associations and pre-prepared lesson plans and assessment resources. There are also many resources online for self-education in general CS and programming, which a subset of teachers may choose to use to extend their own learning, but this is not necessary (and should not be required) for teaching.

Teacher confidence is also crucial for implementing this subject. Teachers can find it hard to navigate unfamiliar terminology, which becomes a barrier to engaging (Munasinghe, Bell and Robins 2021). In a pilot study, Vivian et al. (2020) found that teacher confidence was an issue, particularly for primary school teachers, female teachers and those living away from metropolitan areas. However, they found that after about four years' experience, teachers moved from negative to positive self-esteem in computing, so a key is supporting teachers to overcome initial frustrations as they gain experience. In our experience in New Zealand, an effective way of increasing confidence and engagement for a significant number of teachers is making personal connections, which is a large emphasis in our workshops. These connections are both between teachers, so they can support each other in the future, and between teachers and the subject content itself. Identifying the importance of computing in peoples' lives, and how it connects to their own community, can make it easier for teachers to see themselves as part of the digital world.

While content knowledge is of course essential for teachers, it does not need to be as extensive as might be expected. It's not necessary to be a better programmer than students (although of course more programming knowledge is helpful) and be able to debug their programs; rather it's about facilitating learning and supporting students to do the problem solving themselves.

12.9 Summary: The purpose of teaching computing at primary schools

Now that we have both a big picture of what computing is about and some detailed examples that relate to this, the purpose of teaching these topics in primary schools comes into focus.

A key purpose of education is to prepare students for their future and enable them to participate positively in society. Students will also have the opportunity to see why numeracy and literacy are important if they intend to engage further with computing, so that they don't abandon learning these in favour of just learning 'coding'.

Despite the value of introducing CT at primary school, there are also challenges in the process. Any change in an educational system is complex, and these changes involve a considerable amount of learning for teachers. Making space in a curriculum can also mean a change in priorities that not all education systems are ready for (Brown et al., 2014), and this challenge extends to teacher education.

To make this transition more achievable, we must stay focussed on the main objectives for introducing these topics to primary school curricula. The goal is not to push students' knowledge of these concepts as far as possible. It is to give all students a broad understanding of the topic, teach them the big picture concepts and, for those students who enjoy the topic, unlock their passion for it.

Key points

- Focussing on the big ideas and goals of teaching computing can avoid it turning into a disjoint collection of topics that need to be taught.
- Digital systems are designed for humans, and computing education needs to bring students back to thinking about how each sub-topic studied affects humans.
- Computational thinking can be exercised by teaching skills and knowledge in computing.
- There are several reasons for introducing computing to primary aged students; many of these mean that their learning should ignite their interest rather than overwhelm them with information.
- Integrated learning can help to reinforce the applicability of computing, give teachers more confidence with the subject and avoid displacing other subjects in the school day.
- Learning programming should be seen as gaining a broad understanding of what computing is, rather than a specific skill in a particular language.
- Teachers need support to overcome initial challenges teaching a new topic, as it can take a few years to become comfortable with it.
- Bringing in a new subject to the curriculum is challenging; it is important to keep in mind the main purposes for teaching the new topics.

For further reflection

- How could topics from computing be used to enhance other subjects in the curriculum with integrated learning?
- What might different interest groups (parents, teachers, industry, government) see as the main reasons for introducing computing in primary schools?

References

Aho, A. V. (2010, January), *What Is Computation?* Ubiquity Symposium.

Aivaloglou, E., and Hermans, F. (2016), 'How Kids Code and How We Know: An Exploratory Study on the Scratch Repository', in *Proceedings of the 2016 ACM Conference on International Computing Education Research*, 53–61.

Amanullah, K., and Bell, T. (2019), 'Evaluating the Use of Remixing in Scratch Projects Based on Repertoire, Lines of Code (LoC), and Elementary Patterns', in *2019 IEEE Frontiers in Education Conference (FIE)*, 1–8, IEEE.

Bell, J., and Bell, T. (2018), 'Integrating Computational Thinking with a Music Education Context', *Informatics in Education*, 17 (2): 151–66. https://www.ceeol.com/search/article-detail?id=708739.

Bell, T. (2016), 'Demystifying Coding for Schools—What Are We Actually Trying to Teach?' *Bulletin of the European Association for Computer Science (BEATS)*, 120 (October): 126–34.

Bell, T. (2021), 'CS Unplugged or Coding Classes?' *Communications of the ACM*, 64 (5): 25–7.

Bell, T., Duncan, C., and Atlas, J. (2016), 'Teacher Feedback on Delivering Computational Thinking in Primary School', in *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*, 100–1.

Bell, T., and Vahrenhold, J. (2018), 'CS Unplugged—How Is It Used, and Does It Work?', in H. J. Böckenhauer, D. Komm, and W. Unger (eds), *Adventures between Lower Bounds and Higher Altitudes: Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*, vol. 11011, 497–521, Cham: Springer.

Brown, N. C. C., Sentance, S., Crick, T., and Humphreys, S. (2014), 'Restart: The Resurgence of Computer Science in UK Schools', *Transactions on Computing Education*, 14 (2): 9:1–9:22. http://doi.org/10.1145/2602484.

CS Unplugged (n.d.), Unit Plan: Binary Numbers. https://www.csunplugged.org/en/topics/binary-numbers/unit-plan/.

Duncan, C. (2019), 'Computer Science and Computational Thinking in Primary Schools', Doctoral Dissertation, University of Canterbury, New Zealand. http://dx.doi.org/10.26021/3286.

Duncan, C., and Bell, T. (2015), 'A Pilot Computer Science and Programming Course for Primary School Students', in *Proceedings of the Workshop in Primary and Secondary Computing Education—WiPSCE '15*, New York: ACM Press.

Duncan, C., Bell, T., and Atlas, J. (2017), 'What Do the Teachers Think? Introducing Computational Thinking in the Primary School Curriculum', in *Proceedings of the Nineteenth Australasian Computing Education Conference*, 65–74. http://doi.org/10.1145/3013499.3013506.

Duncan, C., Bell, T., and Tanimoto, S. (2014), 'Should Your 8-Year-Old Learn Coding?' in *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, 60–9, New York: ACM Press. http://doi.org/10.1145/2670757.2670774.

Falkner, K., Vivian, R., and Falkner, N. (2014), 'The Australian Digital Technologies Curriculum: Challenge and Opportunity', in *Proceedings of the Sixteenth Australasian Computing Education Conference (ACE2014),* 3–12, Auckland, New Zealand.

Hermans, F., and Aivaloglou, E. (2017), 'To Scratch or Not to Scratch? A Controlled Experiment Comparing Plugged First and Unplugged First Programming Lessons', in *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*, 49–56, New York: ACM Press.

Lee, I., Martin, F., and Apone, K. (2014), 'Integrating Computational Thinking across the K–8 Curriculum', *ACM Inroads*, 5 (4): 64–71. http://dx.doi.org/10.1145/2684721.2684736.

Munasinghe, B., Bell, T., and Robins, A. (2021), 'Teachers' Understanding of Technical Terms in a Computational Thinking Curriculum', in *Australasian Computing Education Conference*, 106–14.

Resnick, M., and Rusk, N. (2020), 'Coding at a Crossroads', *Communications of the ACM*, 63 (11): 120–7.

Schulte, C., Hornung, M., and Sentance, S. (2012), 'Computer Science at School/CS Teacher Education: Koli Working-Group Report on CS at School, in Proceedings of the 12th International Conference on Computing Education Research: Koli Calling '12, 29–38. Tahko, Finland: ACM. http://doi.org/10.1145/2401796.2401800.

Sentance, S., Waite, J., and Kallia, M. (2019), 'Teaching Computer Programming with PRIMM: A Sociocultural Perspective', *Computer Science Education*, 29 (2–3): 136–76.

Smith, S., and Burrow, L. E. (2016), 'Programming Multimedia Stories in Scratch to Integrate Computational Thinking and Writing with Elementary Students', *Journal of Mathematics Education*, 9 (2): 119–31.

Vivian, R., Quille, K., McGill, M. M., Falkner, K., Sentance, S., Barksdale, S., Busuttil, L., Cole, E., Liebe, C., and Maiorana, F. (2020, June), 'An International Pilot Study of K–12 Teachers' Computer Science Self-Esteem', in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, 117–23.

13

Teaching of Concepts

Paul Curzon, Peter W. McOwan, James Donohue, Seymour Wright and William Marsh

Chapter outline

Chapter synopsis

This chapter reviews strategies successfully used for teaching computer science and computational thinking concepts: those based on analogy, 'unplugged' computing, discourse and writing, contextualized approaches and enquiry-based learning. We give concrete examples based on our practical experience teaching computing concepts to students of all ages through Computer Science for Fun (www.cs4fn.org) and in developing material to support teachers through Teaching London Computing (teachinglondoncomputing.org). We also draw on other resources including CS Unplugged (csunplugged.org), which spawned an interest in kinaesthetic activities stimulating understanding of concepts in concrete ways by making the abstract tangible.

13.1 Introduction

Computing develops sophisticated skills such as programming but is also a rigorous academic subject akin to physics or history, consisting of a rich conceptual framework. Computational thinking, the core skill set that students develop studying computing, is also rich with concepts such as abstraction, decomposition and generalization. Understanding these concepts is an important part of being able to think computationally. A solid understanding of earlier concepts is often a prerequisite for understanding later ones (Meyer and Land, 2006), especially with programming. Having ways to give students timely help around threshold concepts and misconceptions is vital as it is easy for students to form faulty mental models and so harbour critical misconceptions.

The learning of concepts is therefore vital. Appropriate pedagogic methods are needed that leave students with a clear understanding of individual concepts, their relationships and how they fit into broader contexts of the subject. We overview a variety of approaches.

13.2 Teaching through analogy and storytelling

Analogy

Many computing concepts have links to everyday objects and real-world, ideas so the use of analogy is a powerful way to scaffold students' understanding. These analogies can help explain the computing version in a memorable way (see box). Curzon (2014) describes many such analogies.

Example: Analogy for computing concepts

First-in-first-out queue data structures have a direct analogy to what we do in supermarkets. A priority queue might be illustrated by talking about what happens at a night club where most people queue but celebrities go straight to the front, and the arrival of an A-list celebrity leaves minor celebrities waiting.

The idea that one data structure might be implemented in several completely different ways can also be illustrated with real-world examples first. Have students consider different situations they have encountered where queues form (a deli counter and a bus stop, perhaps) and explain how they work. Both are first-in-first-out but may do so with completely different implementations. The bus stop might have people waiting in line: new people join at the back but leave from the front, and all shuffle up as someone leaves. The deli counter might have a ticket system. People take a number and stand anywhere. They move only when their number comes up. Many other implementations exist too, showing that one abstract data type can be implemented in many ways.

It is important to make clear the boundaries of the analogy and clearly link back to the computing concept. This relates to the educational theory of 'semantic waves' (Maton, 2013; Macnaught et al.,

2013), which argues that good teaching involves waves of explanation. First the teacher descends the semantic wave linking from abstract and technical concepts to concrete and everyday concepts. Critically, they then go back up the wave to link back to the technical concepts. Chapter 14 explores the use of metaphors from a linguistic perspective.

Some computing analogies are more than an analogy – the computing and real-world versions are identical at the conceptual level. Computation is something that happens in the world, not just in computers. For example, a stack of chairs follows the rules of a stack data structure in that chairs can be added and removed only in a last-in-first-out manner. It is the implementation of the concepts that differs, not the abstract concept itself. Everyday equivalences make very powerful explanations.

Key concept: Semantic waves

Semantic waves concern what makes a powerful explanation. They explain how to make many of the approaches described here work (or why they do not work). Good explanation starts by introducing technical concepts but then relates them in some way to concrete (or material) situations or contexts that are already understood, before then explicitly linking back to the new concept. This approach is behind successful teaching by analogy, storytelling and successful unplugged teaching.

Storytelling

Storytelling provides a different kind of hook to link concepts to (see box). The stories must be engaging and memorable. The place of the concept in the story must also be natural. The links from the story to the technical concepts have to be clear: travelling the semantic wave. One approach is to tell a story, then have students identify the concepts themselves.

Examples: Storytelling

An example of using fiction to teach concepts is in telling the story of 'The Cat in the Hat' (Dr Seuss, 1958) to explain recursion. In this story a series of ever smaller cats appear to help solve a problem of a stain on a bath. With the cats representing recursive calls, the story includes analogies for the key concepts underpinning recursion: base and step cases, and the need to unwind recursive calls. Linked to the story, these concepts are given a problem-solving context and made memorable.

A non-fiction example is used in the 'Searching to Speak' activity (Curzon and McOwan, 2017; Curzon, n.d.). This uses the story of how a locked-in syndrome, paralysed patient wrote a book. You interactively explore with the class how to devise an algorithm for him to communicate when all he can do is blink an eye. The natural human drama involved, together with a twist at the end, makes the story a memorable way to learn concepts such as divide and conquer.

Invention

Another approach is to support students to *invent* concepts themselves. Puzzles or kinaesthetic activities provide scaffolding. For example, to introduce while loops, introduce the need for repetition and have the students invent the syntax and structure, based on their understanding of if-statements.

13.3 Computing unplugged

Unplugged computing (Bell et al., 2009) involves teaching concepts away from computers. Unplugged activities provide ways to understand concepts in a constructivist way (Papert and Harel, 1991). It is again vital that the links to the concepts are clearly made. Otherwise students can be left understanding the everyday version of the activity but not the computing concept itself. Unplugged activities are a physical, rather than verbal, version of applying semantic waves (Macnaught et al., 2013), where the physical activity is a way of engaging with the concept in a concrete way, before ascending the wave to make the link to the technical concept.

Key concept: Unplugged computing

Unplugged computing involves teaching computing away from the computers. Physical objects are used to illustrate abstract concepts. There are a variety of approaches centred around kinaesthetic activities. Variations include role-playing computation, puzzles, games and magic. The strength is in the way intangible abstract concepts are made physical, so can be pointed to, manipulated and questions easily asked about them.

Kinaesthetic activities

In kinaesthetic unplugged activities (Curzon et al., 2009), tangible objects are representatives of abstract concepts. These can be physically manipulated following algorithms. Learners are able to see, point to and manipulate objects, making it easier to explore the concepts and ask questions. A barrier to asking questions is often that the person does not possess the vocabulary to even frame them. By giving physical representations, the learner can point to them and ask the question at the level of the analogy rather than having to fully verbalize it at the technical level. This can also be used to encourage students to invent concepts.

Example: Kinaesthetic binary search

In the CS Unplugged 'Binary Search' activity (CS Unplugged, 2008), participants invent binary search. A physical scenario for doing the search is set up with

numbered objects placed in order under cups. The student must find a particular object, lifting as few cups as possible. The secret is to check the middle first and use the number there as a signpost indicating which half to then search recursively. If students don't immediately invent this idea, they soon learn that it is a good strategy.

Role-playing computation

The kinaesthetic approach can be taken further: role-playing computation. Instead of physical objects taking the place of virtual things and the student applying an algorithm to them, the computation now acts on the students. The CS Unplugged 'Beat the Clock: Sorting Networks' activity (Bell et al, 2015: 80–6) is one example. Students act as data that is sorted into order following an algorithm.

Example: Role-playing program execution

Assignment is a threshold concept when learning programming. In the Teaching London Computing 'Box Variables' activity (Box Activity Variables, n.d.), students role-play being variables: storage spaces with shredders and photocopiers. Code is executed on them manipulating those variables. This builds a mental model of how a sequence of assignments work. It gives a clear, visual illustration of how a single value is copied from elsewhere, then stored, with old values discarded (shredded). It also makes a tangible distinction between names (a label round the person's neck), the variable (the box the person holds) and the value (the thing in the box).

Students can also role-play commands, wiring them together as in the 'Imp Computer' activity (Imp computer activity, n.d.), to illustrate control structures. 'Compile' program fragments onto the 'wired' together students. Each plays a statement or test, following their instruction only when a baton is passed to them.

Dance has been used to visualize algorithms (Strictly micro:bit, 2016), as has sport, turning sort algorithms into relay races to put numbered beanbags into order in buckets, for example.

Puzzles

Computer scientists have long used puzzles to develop computational thinking skills (Harel and Feldman, 2004; Levitin and Levitin, 2011). Logic puzzles can be used to develop logical thinking skills. Variations of puzzles found in puzzle books can illustrate a wide range of concepts (Puzzles computational thinking, n.d.). They give a simple, but fun, way to actively work with concepts. For example, give ciphertext to decrypt as a puzzle with or without keys. This leads to active learning around concepts including encryption, decryption, frequency analysis, cribs, plaintext and ciphertext, encryption keys, symmetric and asymmetric ciphers as well as different encryption

algorithms. Puzzles can recreate forms of attack such as man-in-the-middle attacks: set the puzzle of forwarding an encrypted message with an additional sentence added. Take a similar approach with other concepts: follow the algorithm to solve the puzzle. For example, compression puzzles are a way to understand concepts around compression of text (Compression code Puzzles, n.d.).

The Swap puzzle (n.d.) involves swapping the positions of pieces by sliding or jumping them. It explores what is meant by an algorithm, how different algorithms can solve the same problem and how alternatives can take different numbers of steps, leading to concepts linked to algorithm efficiency.

Art and craft

Art and craft can be linked to computing concepts. A CS Unplugged activity involves creating necklaces that encode binary in the colours or shapes of beads. Square grid colour-by-number puzzles, where numbers represent the colour of squares, explore concepts related to image representation, introducing bitmap graphics. Interdisciplinary links can be made to Pointilism (painting with small circles) and Roman mosaics (pixels as pieces of glass). Variations illustrate colour depth and image compression. To introduce transmission of images, give students the task of sending a pixel puzzle drawing across the room by holding up numbered placards. Then, have them devise a way of doing it by sending fewer numbers (inventing run-length encoding). Introduce vector graphics using pictures created from shapes in a puzzle context. Introduce recursion using recursive drawings of grass or trees (Doodle Art, n.d.) linking to the computer-generated imagery of films as well as algorithmic art.

Games

Games, from pencil and paper to video games, can teach concepts embedded within them. The game can be centrally linked to the concept as with the card game Control-Alt-Hack® (www.ControlAltHack.com). It explicitly teaches security concepts, with players going on ethical hackers' missions. Other games draw on analogies: use 20-questions activity (n.d.) to illustrate divide and conquer. If you name people in turn, 'Is it Adele?', 'Is it Batman?', and so on, linear searching through the names of famous people, you lose! Playing it well involves finding halving questions. Games can also provide an engaging context as in the 'Brain-in-a-Bag' activity, where unplugged brains play Snap to illustrate neural nets or logic gates (Brain-in-a-bag, n.d.).

Magic and mystery

Magic tricks can illustrate many computing concepts in fun, memorable ways. Challenge the class to work out both how the magic works and the computing link. 'Self-working' tricks of magicians are equivalent to a computer scientist's algorithm. They comprise precise instructions that if followed in the given order always result in a specific, desired magical effect (such as that the card

on the table is the one predicted). Computer programs are just algorithms written in a language that a computer, rather than a human magician, can follow with the guaranteed effect of whatever the program is supposed to do. This analogy gives an interesting way to introduce both algorithms and computational thinking concepts.

Example: A magic trick

Download the teleporting robot jigsaw trick (Teleporting robot (and Melting Snowman) Activity, n.d.). To do the magic, follow the three-step algorithm: count the robots, rebuild the jigsaw in a different way and count the robots again. A robot disappears. Everyone can do the trick by following the instructions without knowing how it works. The point of algorithms is that you need no understanding of what is going on for the correct effect to be guaranteed – that is what computers need as all they do is follow instructions blindly. The magic trick illustrates why algorithms are the basis of computing.

Magic tricks were first used to teach computing as part of CS Unplugged, for example to illustrate error-correcting codes in the Card Flip Magic Activity (Bell et al., 2015: 35–42). 'The Magic of Computer Science' series (McOwan and Curzon, 2008; McOwan, Curzon and Black, 2009; Curzon and McOwan, 2015) take easy-to-perform self-working tricks, showing how the secret techniques can be mapped to many different concepts in computing, from binary to pattern recognition. Tricks work well as openers to a lesson using the emotional hook and mystery of magic to set the scene for in-depth analysis of the taught concepts.

The way tricks are invented gives way to explore computational thinking concepts such as decomposition (full tricks are built from combinations of smaller techniques), generalization (creating a new trick from the principle of an old one) and evaluation (are you sure enough it works to present it to an audience?).

Presentation matters too for a trick to be fit for purpose. This gives a way to introduce concepts around usability, so that programs are fit for purpose. The same understanding of human cognition matters when presenting a magic trick and when designing easy-to-use software (McOwan, Curzon and Black, 2009; Curzon and McOwan, 2015).

Having mastered the performance and learning opportunities of self-working magic, develop your own lessons based on self-working card tricks, starting with Fulves (1976) classic series or Diaconis and Graham (2013) for a more detailed dive into mathematical card magic.

Part of the strength of magic tricks for teaching concepts is the mystery involved, discussed below.

13.4 Context-based learning

Context-based learning involves teaching concepts within either real-life or fictitious, but realistic, examples. This involves actively engaging students with the material rather than presenting concepts in an isolated and theoretical manner. If topics can be set in a context that relates to a

student's interests and pre-existing knowledge and understanding, then that interest can drive their learning. Several contexts can be drawn on to support learning of computing concepts.

Within-syllabus context

The most basic context is that of the wider subject itself. The computing syllabus is not just a series of self-contained areas. There are rich links between them. Drawing out the connections and scaffolding students to do so themselves is important. It is helpful to place more basic topics in a wider context of the subject, so that they are not just abstractions. For example, instead of teaching binary representations in isolation, motivate it by the context of networking and how text is represented, allowing messages to be sent between computers. Concept maps, discussed below, can help students to draw out wider links across a subject. This relates to the higher levels of the SOLO taxonomy, where forming such links is a measure of intellectual progress (Biggs and Collis, 1982).

Technological context

In computing, within-syllabus contexts can be extended to real-world technological contexts. Computers are ubiquitous and disappearing into the environment. Rather than just discuss binary representations of letters and images in the context of networking, put them in the context of instant messaging or texts. Set image representation in the context of image sharing and how a digital camera works. Draw on students' curiosity about how technology actually works.

Historical context

A historical context can support understanding of many concepts and link to history syllabuses. For example, Roman numerals were better for large numbers than the notches in sticks shepherds used. However, it is hard to do multiplication with Roman numerals, so they were eventually replaced. These examples show why number representation matters. Pre-computer-age historical links can intrigue, combining with the storytelling approach of providing strong narratives. Prominence can be given to the roles of women through the history of computing, telling the stories of Ada Lovelace, Grace Hopper and others. The role of Bletchley Park and of Polish Mathematicians, cracking ciphers, gives new understanding about the actions and 'brilliance' of allied commanders and so on.

Example: Historical storytelling

Cryptography and steganography date back thousands of years. Tell the story of the beheading of Mary Queen of Scots. Placed under house arrest by Elizabeth I for twenty-one years, Mary became involved in an assassination plot. She communicated with the plotters by hiding messages in casks of Ale (steganography) and used a simple

substitution cipher. Elizabeth's spy master knew of the plot. His man-in-the middle attack, intercepting messages, using frequency analysis to read them and adding text to the end of messages gave him the evidence he needed to execute the plotters.

Cross-curricular context

Computing has rich interdisciplinary links, drawing on other subjects and changing the way they are done. We have seen links to art and history. Computer scientists have also drawn biology: neural networks (computing based on the way the brain works) and genetic algorithms (computing based on the way evolution works) and the other sciences. Computational modelling, whether unplugged (see, e.g., Brain-in-a-bag activity, n.d.) or programming (see below), can also bring concepts in other subjects to life. At the same time, the concept of computational modelling itself can be explored.

Non-computing context

Non-computing contexts allow students to understand new abstract concepts in terms of things they already understand or that are physical and tangible and so easier to understand first (lower down the semantic wave). Drawing on constructivist principles, it also shows how computer science is about much more than computers. For example, use Braille to introduce binary representations. It is a binary system of bumps and no-bumps that was the first practical use of a binary representation of characters. See also the earlier locked-in syndrome activity.

Research context

Research stories can also be used to embed concepts in context. This is the basis of the 'Computer Science for Fun' approach (www.cs4fn.org). It presents research in a fun way, using accessible language, embedding clear explanations of core concepts in the stories. cs4fn was originally intended as hobby-mode learning to be read as any other magazine. However, schools use it in many other ways including as set reading, as a source of ideas for writing and to support literacy.

13.5 Enquiry-based learning

The way science is traditionally taught is a cause of students' declining interest in STEM with age (Rocard, 2007). Enquiry-based learning, where students explore topics using real scientific methods, helps arrest the decline in student attitudes towards STEM, fosters better scientific thinking and gives a deep way to learn concepts. It can be applied to computing too.

TEMI

'Teaching Enquiry with Mysteries Incorporated' (TEMI; Loziak et al., 2016) is an enquiry-based learning framework using mysteries as a hook to engage students. It defines enquiry in terms of a student cognitive skill set and uses a stepwise progression to engender confident enquirers. It exploits the affective side of learning, the need to engage emotionally and creatively, that is at the core of scientific practice. It is based on the 5Es *learning* cycle (Bybee et al., 2006), an effective methodology for teaching concepts where the students do the work.

Key concept: The 5Es learning cycle

The 5Es *learning* cycle structures lessons around the following five stages.

Engage: Students are first hooked. In TEMI this is by the chosen mystery. It generates questions.

Explore: Students then carry out research and experiments, to answer the question(s) identified in the 'engage' phase.

Explain: Students summarize their learning so far. The teacher assesses understanding, possibly intervening to ensure students correctly grasp principles.

Extend: Students use their new learning in a different context, applying the concepts they have learnt about.

Evaluate: Students reflect on what and how they have learnt so they can build on the skills developed. This stage may include formal testing.

A gradual release of responsibilities through the stages structures the learning activity, starting from the teachers acting as a model, employing a phased apprenticeship approach where students take responsibility for their own learning, mistakes and successes. Showmanship binds TEMI lessons together, maintaining a sense of excitement as activities unfold. See, for example, Loziak, McOwan and Olivotto (2016: 179) for a magic trick example: a number is freely chosen that amazingly divides exactly by numbers given by the magician, engaging the class. In an explore phase, the trick is repeated as the students work out the algorithm involved, followed by an explanation of why it works, based on prime factors. The extend step takes the ideas further into a discussion of prime factors in encryption systems. Evaluation explores if variations work.

Enquiry through programming and computational modelling

If students can program, teach computing concepts with exercises to write programs that implement them. Do this in an enquiry-based way. It forces students to engage deeply with the concepts. Care needs to be taken to scaffold exercises and ensure they are at appropriate levels for students' programming skill.

Computational modelling can, similarly, be used to investigate other subjects and better understand phenomena. The Turtle (Millican, 2016) and Greenfoot (www.greenfoot.org) programming environments come with examples: writing a program that simulates Brownian motion, how ants leave trails or the forces acting on a projectile that lead to its trajectory. This leads to a better understanding of computational modelling itself as an investigative tool.

13.6 Discourse and active writing

Use natural language to directly enhance, and assess, learners' knowledge.

Talking as well as doing

Grover and Pea (2013) argue that students gain a better understanding of concepts if 'discourse-intensive pedagogical practices' are used. This involves combining tasks both with productive teacher-led discussions and with peer discussion (so group work is important). They do this in the context of students using App Inventor to write apps, discussing concepts needed and encountered. Chapter 15 further explores the role of classroom discourse in teaching and learning.

It is also part of the exercise to encourage students to look actively for computing concepts in everyday examples. Furthermore, encouraging students to always explain their programs (or algorithms) to someone else (and in documentation) is a powerful way for them to gain a deeper understanding of the concepts. It is also a very positive part of debugging. Often bugs highlight incorrect mental models of underlying concepts. Trying to explain how a program works can lead to the student seeing the problem themselves.

This links to semantic waves (Macnaught et al., 2013): the teacher encourages discussion moving from concrete activities to concepts.

Short writing activities

Defining and explaining concepts, or exploiting concepts when explaining code, are common teaching strategies. Asking learners to do the same is a common form of assessment. The short texts which learners *write* in response give useful snapshots of their knowledge of the concepts concerned. Although a piece of writing can seem to be a static and fixed product, it is the embodiment of lots of processes of thinking and composition. Focussing on how a text is composed is a good way to make a student aware of the thinking – that is, the conceptualization – embodied in that writing.

Example: Short writing activity

This activity revises conceptual knowledge and focusses attention on how to present it effectively in an assessment context. Students involved commented they enjoyed it because it 'was testing our understanding, not just telling us things'.

It focusses on 'what makes a good exam answer' to a question such as 'Explain what is meant by a "system call" '. Review the concepts involved, then have students discuss their expectations about the answer in small groups. Next, provide five answers to this question that range from high scoring to low scoring. Students role-play being examiners in groups, grading the answers using terms like poor, some merit and good or by putting them in order. The class vote on grades before the actual grades are shared. Next, the groups discuss the 'criteria' they used. If they generate criteria specific to the particular question, steer them towards more generic criteria. Have them share their thoughts with the class, before sharing the criteria used by the examiners, part of which involves correct concepts and relationships between concepts and part is to do with the command words (such as 'Explain' or 'Compare') in the question and how these predict the shape of the answer.

As a follow-up, take students' answers where some of the statements are related to the question (if not accurately) and some are not. Present the answers broken into their separate statements. Students work in groups to identify two statements which deal with concepts relevant to the conceptual field of the question and two that are irrelevant, belonging to different conceptual fields. Alternatively, provide lists of concepts which are related to a question. Students allocate the concepts to columns in a table headed: Most Related, Somewhat Related, Little Related. Share the students' tables and invite comments.

Understanding concepts involves understanding relationships between concepts. Explain this, then present students with sentences linking concepts in which one of the concepts is omitted (e.g. 'A system call causes a _____'), together with a mixed-up list of the omitted concept words. In groups they complete the sentences, then discuss.

Concept maps

Concept maps give a visual way to work actively with concepts and the relationships between them. Use them to develop understanding and as a diagnostic test. They give a way for students to frame their thoughts and see when those thoughts are still hazy. They can also use them as a stepping stone towards writing sentences. One way to use concept maps is as a variation on writing relationship statements. Partially created concept maps must be filled in using a set of given terms. Another is to draw concept maps based on the students' own written answers to questions, comparing these with an expert concept map. Do this as part of a role-playing exercise where the students play examiners, commenting on the work of others based on discrepancies to the expert concept map.

Diagnostically, use concept maps to identify students who need extra support in an area or to correct their misunderstandings face to face. This role of concept mapping as a formative assessment method is highlighted in Chapter 16. Another variation is to give students a question and first brainstorm around twenty concepts that are related to it, before creating a draft concept map using them. Also encourage students to draw links between concept maps of different conceptual areas. By doing concept map activities over time, students start to build much richer conceptualizations of the subject themselves (Novak and Canas, 2006).

Program and algorithm comprehension

Programming is a skill. However, it is still important that a strong focus is placed on the deep understanding of underlying concepts. It is easy to focus on syntax, suggesting the task of learning to program is one of remembering keywords and punctuation. Teaching approaches need to place the focus on semantics instead. That involves deeply understanding concepts. The many ways to teach concepts already discussed provide ways to do this, but with programming, formative assessment based on *dry run* activities is critically important. Dry runs involve stepping through the execution of a program on paper, keeping track of the values in variables and the results of tests as each line of a program executes. They help students understand the step-by-step workings of the program and so of the concepts behind the underlying constructs. The focus of dry run activities, and of feedback around them, should not be on the final answer but on the program's step-by-step operation. Integrate pencil and paper dry run exercises with drama-based unplugged techniques, acting out the execution code to help demonstrate constructs and concepts.

13.7 Conclusions

It is easy to fall into the trap of believing that teaching concepts boils down to giving clear definitions for students to learn. This is a transmission model of teaching. We have given many alternative approaches that allow learners to directly engage with concepts in more constructivist ways, building on their existing knowledge and experience. The best approaches combine these techniques giving multiple and rich ways to understand concepts and their interrelations. The ideas behind semantic waves are key to many of these approaches. They involve in various ways travelling the semantic wave, from technical concept down to everyday experience, but then critically repacking back up the wave, making clear links back to the technical concept.

Key points

- Success in programming, computational thinking and computing more generally is founded on a deep understanding of rich concepts.
- Concepts do not have to be taught using a transmission model of teaching.
- There are a wide variety of constructivist approaches that can be used: from analogy-based and unplugged approaches to those built around discourse and active writing.
- Aim to embed discussion within one of the many contexts available for teaching computing concepts.
- Tell stories and use mystery to engage pupils with concepts in memorable ways.
- Use a mixture of approaches and contexts.
- Whatever techniques you use to teach concepts, aim to make the students travel up and down the semantic wave.

For further reflection

- Consider how you teach concepts currently. Do you follow a semantic wave, starting from abstract technical concept, moving to concrete everyday experience and then explicitly linking back to the abstract concept again?
- Identify concepts you currently teach using transmission and explore which approaches apply.
- Identify threshold concepts that students struggle with. Could a targeted change help students overcome the problem?

References

Algorithmic Doodle Art (n.d.), https://teachinglondoncomputing.org/algorithmic-doodle-art/.

Bell, T., Alexander, J., Freeman, I., and Grimley, M. (2009), 'Computer Science Unplugged: School Students Doing Real Computing without Computers', *New Zealand Journal of Applied Computing and Information Technology*, 13 (1): 20–9.

Bell, T., Witten, I. H., Fellows, M., Adams, R., McKenzie, J., Powell, M., and Jarman, S. (2015), CS Unplugged. http://csunplugged.org/wp-content/uploads/2015/03/CSUnplugged_OS_2015_v3.1.pdf.

Biggs, J. B., and Collis, K. (1982), *Evaluating the Quality of Learning: The SOLO Taxonomy*, New York: Academic Press.

The Box Variable Activity (n.d.), https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-box-variable-activity/.

The Box Variable Activity (n.d.), https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-box-variable-activity/.

Bybee, R. W., Taylor, J. A., Gardiner, A., van Scotter, P., Powell, J. C., Westbrook, A., and Landes, N. (2006), The BSCS 5E Instructional Model: Origins and Effectiveness. http://science.education.nih.gov/houseofreps.nsf/b82d55fa138783c2852572c9004f5566/$FILE/Appendix%20D.pdf.

Compression Code Puzzles (n.d.), https://teachinglondoncomputing.org/compression-code-puzzles/.

CS Unplugged (2008), Unplugged: The Show. Part 10: Binary Search – Divide and Conquer (video file). https://www.youtube.com/watch?v=iDVH3oCTc2c.

Curzon, P. (n.d.), Computational Thinking: Searching to Speak. https://teachinglondoncomputing.org/resources/inspiring-computing-stories/computational-thinking-searching-to-speak/.

Curzon, P. (2014, February), Computing without Computers, V 0.15, QMUL. https://teachinglondoncomputing.org/resources/inspiring-computing-stories/computingwithoutcomputers/.

Curzon, P., and McOwan, P. W. (2015), The Magic of Computer Science III: Magic Meets Mistakes, Machines and Medicine, QMUL. https://cs4fndownloads.wordpress.com/magic-computer-science-3/.

Curzon, P., and McOwan, P. W. (2017), *The Power of Computational Thinking: Games, Magic and Puzzles to Help You Become a Computational Thinker*, Singapore: World Scientific.

Curzon, P., McOwan, P. W., Cutts, Q. I., and Bell, T. (2009, July), 'Enthusing and Inspiring with Reusable Kinaesthetic Activities', *ACM SIGCSE Bulletin–ITiCSE '09*, 41 (3): 94–8. https://doi.org/10.1145/1595496.1562911.

Diaconis, P., and Graham, R. (2013), *Magical Mathematics*, Princeton, NJ: Princeton University Press.
Dr Seuss (1958), *The Cat in the Hat Comes Back*, New York: Random House.
Fulves, K. (1976), *Self-Working Card Tricks*, New York: Dover.
Grover, S., and Pea, R. (2013), 'Using a Discourse-Intensive Pedagogy and Android's App Inventor for Introducing Computational Concepts to Middle School Students', in *Proceedings of the 44th SIGCSE Technical Symposium on Computer Science Education*, 723–8, ACM. https://doi.org/10.1145/2445196.2445404.
Harel, D., and Feldman, Y. (2004), *Algorithmics: The Spirit of Computing*, Boston, MA: Addison Wesley.
The Imp Computer Activity (n.d.), https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-imp-computer-activity/.
Levitin, A., and Levitin, M. (2011), *Algorithmic Puzzles*, Oxford: Oxford University Press.
Loziak, D., McOwan, P. W., and Olivotto, C. (eds) (2016), The Book of Science Mysteries, Version 2.0, TEMI. https://cs4fndownloads.wordpress.com/temi-book-of-science-mysteries/.
Macnaught, L., Maton, K., Martin, J. R., and Matruglio, E. (2013), 'Jointly Constructing Semantic Waves: Implications for Teacher Training', *Linguistics and Education*, 24: 50–63. https://doi.org/10.1016/j.linged.2012.11.008.
Maton, K. (2013), 'Making Semantic Waves: A Key to Cumulative Knowledge-Building', *Linguistics and Education*, 24 (1): 8–22. http://doi.org/10.1016/j.linged.2012.11.005.
McOwan, P. W., and Curzon, P. (2008), The Magic of Computer Science: Card Tricks Special, QMUL. https://cs4fndownloads.wordpress.com/magic-computer-science-1/.
McOwan, P. W., Curzon, P., and Black, J. (2009), The Magic of Computer Science II: Now We Have Your Attention …, QMUL. https://cs4fndownloads.wordpress.com/magic-computer-science-2/.
Meyer, J. H. F., and Land, R. (eds) (2006), *Overcoming Barriers to Student Understanding: Threshold Concepts and Troublesome Knowledge*, London: Routledge.
Millican, P. (2016), *The Turtle System: Computer Science across the Curriculum*. http://www.turtle.ox.ac.uk/csac.
Novak, J. D., and Cañas, A. J. (2006), *The Theory Underlying Concept Maps and How to Construct Them*, technical report No. IHMC CmapTools 2006–1, Pensacola, FL: Institute for Human and Machine Cognition.
Papert, S., and Harel, I. (1991), *Constructionism*. New York: Ablex.
Pixel Puzzle Pictures and Computational Thinking (n.d.), https://teachinglondoncomputing.org/pixel-puzzles/.
Puzzles and Computational Thinking (n.d.), https://teachinglondoncomputing.org/puzzles/.
Rocard, M. (2007), *Science Education NOW: A Renewed Pedagogy for the Future of Europe*, Brussels: European Commission. http://ec.europa.eu/research/science-society/document_library/pdf_06/report-rocard-onscience-education_en.pd.
Strictly micro:bit—Live Lesson (2016), http://www.bbc.co.uk/programmes/articles/49tjW0qR05wXrdpK7ZbGTbs/strictly-micro-bit-live-lesson.
The Swap Puzzle Activity (n.d.), https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-swap-puzzle-activity/.
The Teleporting Robot (and Melting Snowman) Activity (n.d.), https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-teleporting-robot-activity/.
The 20-Questions Activity (n.d.), https://teachinglondoncomputing.org/resources/inspiring-unplugged-classroom-activities/the-20-questions-activity/.

14

Language and Computing

Ira Diethelm, Juliana Goschler, Timo Arnken and Sue Sentance

Chapter outline

Chapter synopsis

Spoken and written language are an important part of the computer science classroom yet often not given much attention. In this chapter we focus on instructional language in the computer science classroom and also address other aspects of language and literacy development through computer science. We introduce some aspects of theory related to this problem domain to start a meta-discourse on spoken language for teaching CS and consider different types of teacher and student talk in the classroom. We give hints and pedagogical suggestions for teachers on how to support understanding and learning computer science as well as supporting literacy development in students.

14.1 Introduction

Whenever you learn a new subject or skill, at some point you need to pick up the particular language that goes with that domain. And the only way to really feel comfortable with this language is to practice using it. It's exactly the same when learning computer science (CS). In this chapter we consider how language is used in the CS classroom, in terms of both the terminology of the subject and the way we can use dialogue and questioning effectively.

14.2 Language in education

In any subject, a learner needs to know about the terminology of the subject of interest. But it is important to not only know the terminology but also how to use it in a given context. Certain words may have context-specific meanings that change depending on the subject at hand. Certain fields may have a higher number of metaphors. Or there may be numerous words from a field that found their way into the vernacular of the people but with slightly changed meanings. These specifics need to be known in order to communicate successfully about them and to exchange and expand knowledge.

Successful communication about digital media, computers and CS requires certain linguistic skills. Thus, learning about digital media and computers is closely connected to the ability to understand and produce appropriate descriptions and explanations of the subject matter (Diethelm and Goschler, 2014). Linguistic competence, therefore, is key to successful acquisition of knowledge and learning of skills (Dawes, 2004). This is especially the case in schools, where most instruction is delivered verbally – in the form of oral explanations by the teacher and written texts in textbooks.

Key concept: Educational language

The variety of language used in schools – in the German context often described as 'Bildungssprache' (educational language) – differs considerably from the one used in everyday communication: it is usually less dialogic, more abstract; it uses more complex constructions and in general is more oriented towards written discourse instead of spoken discourse (even if it is delivered verbally). In order to master communication within this variety, one needs more and other linguistic and communicative skills than necessary for everyday interaction outside of educational contexts.

Language relating to a specific discipline may be known as 'disciplinary literacy'. This refers to the subject-specific terminology to which teachers need to introduce students.

When considering the different communication skills, we can make a distinction between basic interpersonal communication skills (BICS) and cognitive academic language proficiency (CALP) (Cummins, 1979, 2008). BICS are sufficient to manage simple 'conversational' language used in

direct everyday communication outside of professional or educational contexts. This language typically consists of dialogue including small talk, simple orders and requests, simple narrations and the like. CALP, in contrast, is necessary in order to master educational and academic language. This language used in schools, universities or professional contexts includes different text types like news, reports, scientific papers, talks or more complex narrative texts like novels. CALP therefore requires – among a lot of pragmatic knowledge – a larger lexicon with more abstract terminology as well as the mastery of certain complex grammatical constructions like sentences in the passive voice, complex noun phrases, complex sentences often including multiple embedding and in some languages such as German other inflectional forms (like past forms of verbs).

Not all pupils are sufficiently equipped with this linguistic knowledge. Many pupils have not had enough input in the required variety, because they do not talk much about topics outside of everyday interest in their family and their peer groups, because they do not read much and/or because they do not engage much in activities that make complex and abstract language necessary. The situation can be even worse for children and youths growing up with more than one language: For many of them, even a large part of everyday communication takes place in another language than in the one they have to use in school, so that their input in the language used in school is even more diminished. But even if bi- or multilingual pupils have a lot of academic and educational input, using their second language often means a higher cognitive load and requires greater concentration on the processing and production of language.

14.3 Language in computing education

It is clear that every subject comes with a specific terminology that has to be learned along with the concepts connected to them. If the terminology consists of words that are used only in connection with the subject, then most of the time teachers are aware of the fact that these words have to be part of their explicit teaching. In CS, words like byte, integer, compiler and recursion and acronyms like RAM/ROM or CAD/CAM have to be taught explicitly.

Example: Creating a glossary

Ask students to write down scientific terms and words that are connected to the present topic of your course and collect them all. Then, let your students create a glossary for the twenty or thirty most important of these words: Therefore, each student should provide a description for one term in his or her own words. Then, two other students should review and adjust each description. Repeat this for each new topic of your course at the beginning of a unit or for preparing for a test.

For other words, it is more difficult for teachers to detect potential challenges. This is often the case with words that have a specific meaning within the CS community but a slightly or sometimes completely different meaning in everyday discourse, for example nouns like code, memory, address,

folder, loop, bug and value or verbs like to save, to code, to retrieve and to submit. Very often, teachers are not aware of the fact that these differing meanings have to be taught and learnt just like completely new words. If this explicit teaching/learning does not take place, pupils try to make sense of the words and the contexts they occur in with their knowledge from everyday communication. This can lead to misunderstandings and misconceptions not helpful for understanding new concepts taught in computer science. Even teachers' questions aiming to make sure that everything is understood often fail to yield appropriate responses by the pupils, because they think they 'understand' all the words that were used, because they 'know' them – albeit with a different meaning.

In the context of German schools, the fact that a lot of CS terminology consists of English words makes the situation different from that in English-speaking countries. On the one hand, having to learn and use a variety of words in another language than the one the subject is taught in could be an additional difficulty for pupils. On the other hand, it makes terminology more perceivable and distinguishable as terminology (with a different meaning than the same word in everyday discourse). In cases where there are German words, however, the problem of different usage in different discourses and communities remains – for example with terms like programmieren (to program/to code) or ein Programm schreiben (write code/write a program).

This terminology is not used in isolation, however. The terms appear in sentences whose construction differs from that of sentences in the vernacular. Most scientific communities and the texts produced within these communities use very specific linguistic constructions that can be rare in everyday discourse or – just like terminology – come with subtly different meanings. Sometimes these differences are rather obvious and easy to figure out: if a teacher orders his or her pupils to 'tell' the computer something, in most cases the pupils would understand that they are meant to type something in, not to say something. However, in other cases it might be less obvious and produce misunderstandings: If a teacher wants his or her students to print a sentence, they might assume that they have to use the printer. But in programming, printing a sentence usually refers to displaying a sentence on screen.

In more advanced CS lessons, the use of highly specific constructions could lead to situations where not everything is understood by the pupils. For example, consider the amount of technical language involved in the task to 'create two derived classes that inherit the properties of the base class and override two inherited methods, but also remember to regulate access through access control attributes'.

14.4 Talk in computing lessons

The nature of student and teacher talk is a key aspect of the teaching of computing in school. However, this has not been widely explored within computing education, although there is a significant body of work in mathematics, science and general education that relates to enhancing the quality of classroom talk (Sentance and Waite, 2021). In computing education, most of the literature relate to language and communication as a vehicle for learning centres on pair programming and peer instruction (Vahrenhold, Cutts and Falkner, 2019), both privileging classroom talk and purposeful dialogue. Research has shown that peer instruction positively impacts learning outcomes (Porter

et al., 2011; Zingaro et al., 2014). Pair programming has been shown to improve program quality and confidence (Braught, Eby and Wahls, 2008; McDowell et al., 2006), although in the school context it may depend on the way that the collaborative work is instantiated (Lewis, 2011).

Encouraging more productive talk in the classroom means supporting students in learning to reason effectively and to explain clearly. In computing, students particularly need such linguistic skills in programming, in order to explain how their program works (or why it doesn't) (Sentance and Waite, 2021). A number of models of classroom dialogue have been proposed to understand how students and teachers can use talk more effectively, including dialogically organized instruction (Nystrand et al., 2003), exploratory talk (Mercer, 1995) and dialogic teaching (Alexander, 2006).

Key concept: Exploratory talk

The notion of exploratory talk (Mercer, 1995) led to a large body of research in mathematics education aimed at improving students' abilities to learn more effectively through improving their talk within a discipline. Exploratory talk is where pupils listen critically but constructively to each other's ideas. The objective of this interaction is to reach an agreement. In the task, students must explore the different possible answers. They exchange ideas with a view to sharing information to solve problems.

Cui and Teo (2020) provide a useful summary and synthesis of the research on dialogue in the classroom and describe a number of 'dialogic moves' that teachers use in the classroom including

- Eliciting a contribution such as through authentic questioning
- Extending dialogue by asking learners to explain through elaboration or substantiation
- Connecting links between participants and their contributions
- Challenging participants to clarify and deepen thinking
- Critiquing through critical evaluation of each other's contributions

Most of these involve skilful questioning on the part of teachers in order to encourage students to use computing-specific language and reasoning to explain their understanding. This is in contrast to typical IRF (initiation–response–feedback) exchanges where a teacher asks a question to which they already know the answer.

Questioning in programming lessons is particularly important in terms of facilitating code comprehension. Questions structured around the Block Model (Schulte, 2008) can be used to ensure different questions target understanding of different elements of the code (Sentance and Waite, 2021).

Some recent research has been conducted specifically on talk in the computing classroom. Sentance and Waite (2021) synthesized discourse frameworks associated with the study of talk in general teaching and learning to analyse talk in high-school programming classrooms where the PRIMM pedagogy was being used. The authors developed a generic theoretical model for planning and evaluating talk in the programming classroom (see Figure 14.1) and found several key factors that enhanced discourse. Key factors included encouraging talk through classroom

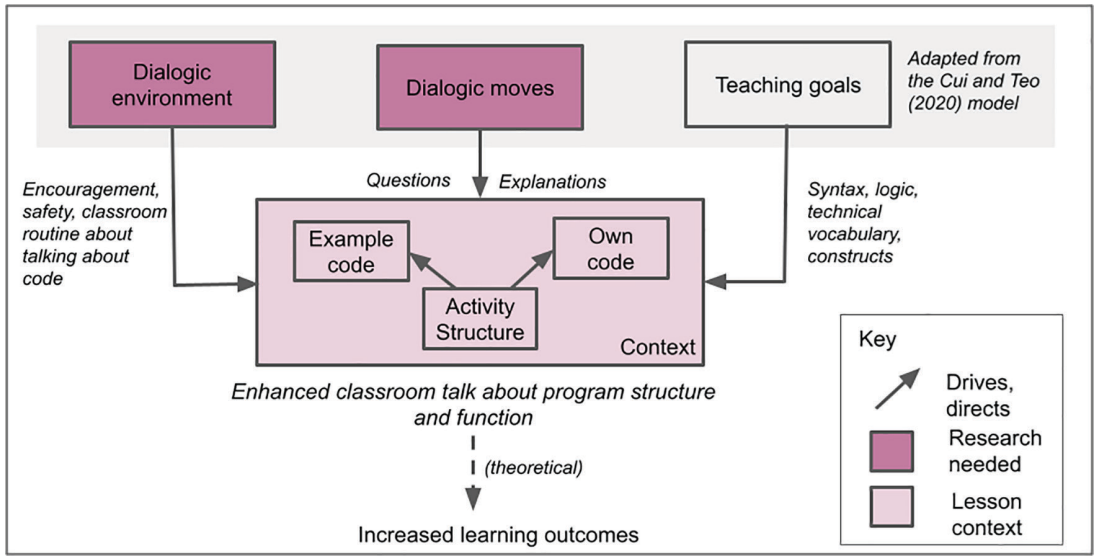

Figure 14.1 Talk in the programming classroom (*Source:* Sentance and Waite, 2021: 13)

routines, using questions and explanations, including goals on vocabulary and carefully designing learning contexts, including using example code, activity structure and the student's own code to situate talk (Sentance and Waite, 2021).

In another study, Zakaria et al. have designed and investigated a structured feedback intervention for teachers to use to support students doing shared programming tasks (Zakaria et al., 2021). Comparing the dialogue and activity of six pairs of students, aged ten to eleven years old, from classes with and without the intervention, the authors reported promising results in productive collaboration and discourse such as increased exploratory talk including more justification and an increase in shared alternative ideas. Research on classroom talk in computing is still at an early stage, and it is clear that more is needed.

14.5 Metaphors of computer science

Metaphors are not just fancy additions to our language – more often than not, they reflect certain conceptions and help us to understand abstract things by transferring our knowledge about more concrete things to the abstract domains. This is also the case for computers and things connected to them. It is not unusual to describe computers as containers, sometimes specifically buildings, where things on the inside are moved around, for example in expressions as follows:

- The data is transferred.
- I will move the file to another folder.

More typical words reflecting this metaphor are identified in the study by Izwaini (2003: 3): architecture, library, sign in/log in, sign out/log out, platform, port and window.

The metaphorical description of the computer or parts of it as a living being or even a person is also frequent:

- The computer memorizes previous activities.
- The computer feels asleep/woke-up.
- The compiler looks into the memory address.

Izwaini (2003: 2) identifies more words used in talking about computers that imply that it is a living being: 'client, conflict, dialogue (conversation between the computer and the user), generation, language, memory, protocol, syntax, widow/orphan, and virus and bug (it can get ill)'.

Other metaphors suggest that the computer is a workshop or a manufacturer (equipment, hardware, install, load, template, tools) or an office (desktop, directory, document, file, folder, mail, trash, can, wastebasket) (Izwaini, 2003: 2).

It has become clear that complex and subtle knowledge about the specific use of language within a scientific or educational community is necessary in order to properly understand what is said in the classroom or written in textbooks. It has also become clear that one cannot expect pupils to come readily equipped with all that knowledge, but that teaching specific linguistic knowledge and skills is in fact part of education – thus, subject teaching and language teaching, as well as subject learning and language learning, are inseparable.

14.6 From everyday language to scientific language

So far, we have seen how metaphors are used when talking about CS and quickly touched upon the importance of avoiding misconceptions (see also Chapter 13). One of the problems that arises when teaching CS is ambiguity. In general, CS uses a variety of 'dead metaphors'. These metaphors are not only used, they are 'lived by' (Lakoff and Johnson, 2003). Words like 'packet', 'string', 'protocol', 'cloud', 'stack', 'program', 'model' and many more are not used metaphorically anymore. These ambiguous terms have found their way into the general terminology of the field and are now considered scientific terms (Diethelm and Goschler, 2014: 3). Their meanings shift, depending on who uses one of these words and in which context they are used. In CS lessons, there are also anthropomorphised metaphors. These are metaphors where comparisons with the human body are made. A computer works like a brain; it can break down, can get a virus and needs care (Steffen, 2006: 42), processes can be killed and there is inheritance in programming.

Key concept: Metaphors

Metaphors can help pupils and teachers to conceptualize very abstract things and make them cognitively more manageable. However, it always has to be clear that these are in fact metaphors, not factual descriptions of the parts and actions of the computer.

Some metaphors can be helpful in some respects and on certain levels of description – for example the distinction between programs and folders – but they can be misleading in other respects and on other levels – for example if one wants to explain the basic nature of 'information' the computer uses. In order to avoid long-lasting misconceptions, which can result in resistance to learning new conceptualizations, teachers have to be aware of the metaphors used in classroom language, in order to avoid them or use alternative ones if a specific metaphor blocks the understanding of a certain concept. We also suggest that teachers should explain carefully rather than penalize the usage of anthropomorphised metaphors. After all, even professionals tend to use these kinds of metaphors (Anton, 2010: 68). Using metaphors leads to discourse about metaphors, which encourages more dialogue about the subject between students and teachers.

In his PhD thesis, Busch researched the usage of metaphors in CS (Busch, 1998). This allows for a first detailed look at the way metaphors are used in CS in general and the way they are used in teaching in particular (Diethelm and Goschler, 2014). There, he provides a three-step system, which can help teachers if they want to use metaphors:

1. Clarification of meaning of the term in other non-CS contexts
2. Speculation about possible meanings of that term in CS and also about what it would certainly not mean
3. Development of the CS-related meaning of the term (Busch, 1998: 125)

These three steps can be used by teachers to make sure that the origin as well as the meaning of a metaphor is made clear to the students, that the way the metaphor is used by the students is correct and that the students can distinguish between metaphors for everyday language and those used in technical contexts.

Teachers can also make these distinctions by clarifying the level of language that is used in classroom settings. Given that there are at least two different levels of language, everyday language and scientific language, teachers therefore need to make sure that there is time to talk about (scientific) language and its benefits in the classroom.

In order to introduce new scientific language to the classroom, Martin Wagenschein (1980: 130–8) suggests three phases. The first phase aims at causing surprise. Students are encouraged to express their feelings and thoughts in their own words. Every utterance is valuable to the discourse. The teacher needs to take a step back and only guide the discussion using non-scientific language. The second phase aims at preserving these thoughts by embedding them into everyday language in a way 'that it could be explained to oneself or others in a way that it could be understood later on, e.g. in a year' (Wagenschein, 1980: 130–8). So far, only common language is being used. This changes in phase three. The third phase uses the preserved text and transforms it into a scientific text. Wagenschein suggests that scientific language should not be taught explicitly. We, on the other hand, suggest that the transformation in phase three should be taught explicitly to make sure that students understand that the terminology used is different from everyday language. This way, inaccuracies and misconceptions can be avoided and clearer lines between the different levels of language can be drawn. A way for teachers to introduce new language explicitly in phase three is to incorporate a meta-discourse in their lessons.

Key concept: Meta-discourse

The meta-discourse's aim is 'to engage students in a discussion about language including syntactic and semantic features of informal everyday talk and of formal scientific use' (Rincke, 2011). Teachers can single out the settings in which scientific words or phrases can occur, make distinctions between the ways they are used in common language and scientific language, make cross-disciplinary comparisons and show how words and sentences surrounding the word in question shape meaning.

Using a meta-discourse allows students not only to understand where a word or a phrase comes from and what it means but also to assign it to a level of language. It is then easier for students to distinguish between the levels of language. In his book, Lemke suggests that the ability to talk about science also leads to the ability to do science 'in terms of reasoning, observing, analysing and writing' (Diethelm and Goschler, 2014: 3). Lemke (1990: 170) also recommends that students 'should be required to be able to say anything in science in more than one way, and taught how to do so. … Saying the meaning without the same set of words'.

14.7 Structuring lessons

When structuring lessons, teachers need to be aware of possible difficulties when introducing new terminology. A common model used to structure student-oriented lessons is Educational Reconstruction. This model can also be applied to plan a language-oriented lesson including a meta-discourse. The model places a phenomenon caused by CS in its centre and surrounds it by five aspects that all interact with and influence each other (see Figure 14.2). Each of these components can be used for student- and language-oriented lessons.

The first aspect is the *analysis of social demands*. This aspect focuses on explaining why a certain concept or phenomenon needs to be taught. What makes it important? Why should students be able to talk about this concept? When a phenomenon is being chosen as the topic for a lesson, it is not only important to choose a topic that the students experience outside of their classrooms but the teacher also needs to ask questions regarding the origin of the phenomenon. What is its name and where does its name come from? And is it necessary or helpful for the students that they get this information as well? Essentially, the teacher needs to decide if the phenomenon itself needs a linguistic analysis in class or not. Even if the phenomenon itself does not need explicit linguistic coverage, its parts just might. The Clarification of Science Content Structure 'describes how the phenomenon could be explained scientifically and which subject domain knowledge is required to understand the phenomenon' (Diethelm and Goschler, 2014). Kattmann et al. list several questions for this step, one of which is especially important for a language-oriented lesson: 'Which terms are used in scientific publications and textbooks and which of them could hinder or support learning due to the narrow meaning of the word or parts of it?' (qtd. in Diethelm and Goschler, 2014). It is the teacher's task to choose which terms are important for the lesson, which terms need to be addressed

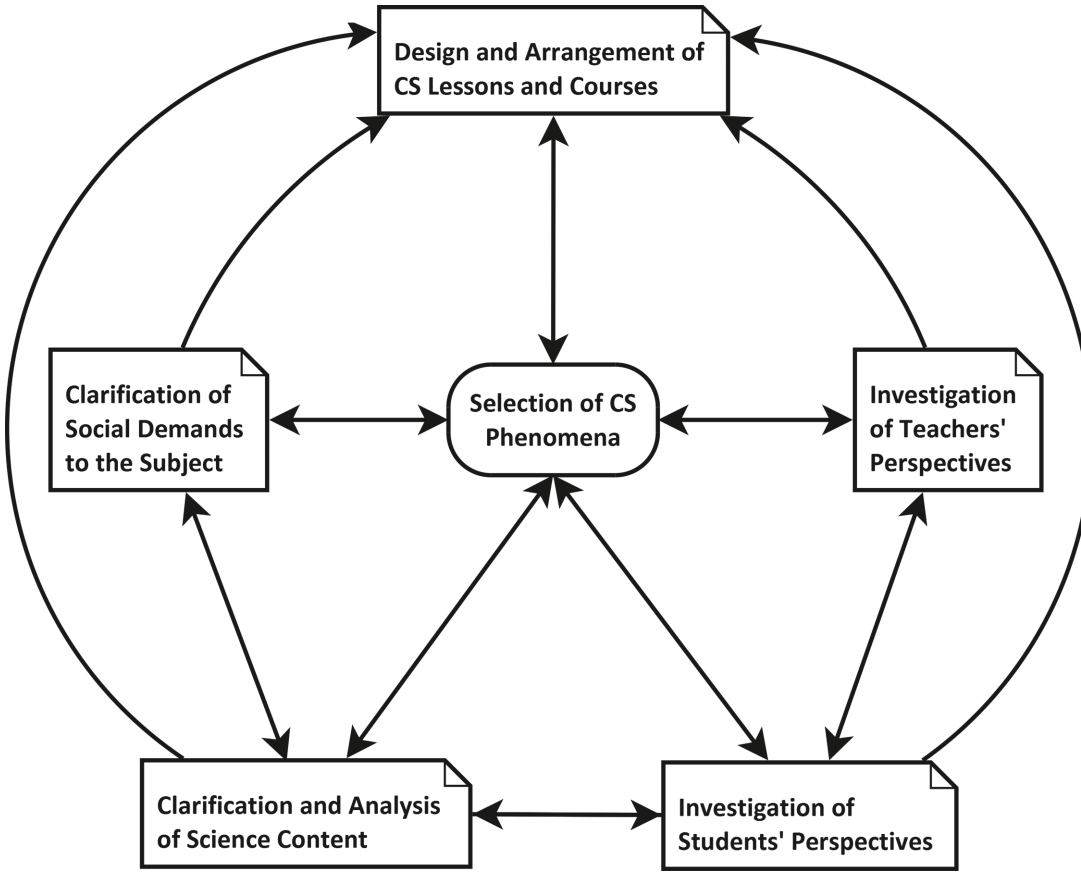

Figure 14.2 Educational reconstruction for computing education (*Source:* Diethelm et al., 2012: 166)

linguistically and how these terms are defined. Teachers need to make sure that new scientific terms are not defined by other scientific terms unless unavoidable. In addition, definitions should be phrased in a way that they are extendable once a deeper understanding of the topic is achieved.

While analysing the *content structure* it is also important to consider both the students' and the teachers' perspectives. When considering the student's perspectives, it is important for teachers to understand that these perspectives matter just as much as the content structure. It is important to understand that students' conceptions are not synonymous with misconceptions, because the 'term "mis" already suggests that the students' conceptions would be wrong or not worthwhile and thus have to be replaced in their minds by something that is more "correct" in any respect' (Diethelm, Hubwieser and Klaus, 2012). It is important to understand the terms students use and why they use them, because they provide insights into the students' lives and enable establishing connections and expanding their knowledge. Teachers need to ask themselves where their students' conceptions derive from, if they derive from linguistic phenomena and also if a meta-discourse can help to create awareness.

Example: How a streaming video works

A study by Diethelm and Zumbrägel (2010) has shown what some pupils think about the way streaming a video works. In their study, 26 per cent of students thought that the video they stream is actually played online and they just watch it, similar to watching a movie in the cinema or on television. While planning a lesson, teachers can then make time to analyse the word 'streaming' together with the children, so that they understand that data are actually transported to their computers, much like water in a stream. Of course, depending on the grade one is teaching, the explanation can be more or less detailed. It can also be used to introduce a new topic, for example the internet, packaging or protocols. Knowing the students' perspectives shapes the way the content structure is presented. Adjustments can be made so that these perspectives are incorporated, problematized and expended in the lesson.

Teachers' perspectives can, just like the student's perspectives, be very different from one another. Teachers will use different methods and have different pedagogical knowledge, different content knowledge, different psychological knowledge and so on (Diethelm, Hubwieser and Klaus, 2012). Therefore, teachers could ask themselves a set of questions when planning a lesson, for example which terms are known to them and how they would explain these terms to their students. This question will also reveal if there are uncertainties of a certain topic. They have to ask themselves which terms are known to them through their work or studies and if there are limits to their own knowledge and where they need to catch up to be able to properly explain concepts to their students. At the same time, CS teachers can also talk to their colleagues or CS teachers from other schools. That way, they can compare definitions for terms among each other. If one supposes that a class has several teachers throughout their years at a school, one can also suppose that the students will encounter teachers who use different definitions for the same terms. Through the comparison of terms, teachers can create a common understanding and use the same terms in the same ways, so that the students can continue to use the terms they learned without causing misunderstandings.

One way to make this common ground available for present as well as future teachers is through the implementation of a wiki. Our work with teachers has shown us that, given a specific concept, different teachers come up with different definitions. These definitions aim at explaining the same concept using different terms, which in turn leads to lively discussions. Through these discussions, teachers experience different points of view and in the end find similar definitions.

All of these aspects influence the design and arrangement of lessons and courses. 'If we focus on "CS classroom language", this means that they all have an influence on the terms that occur in class during different phases of the lessons, used by different participants' (Diethelm and Goschler, 2014). Through iterative execution of the analysis of these aspects, lessons can be planned that are

student- as well as language-oriented and that create an awareness for the language and terminology that is being used throughout the lessons.

14.8 Recommendations

As a result of our thoughts presented so far, and based on the works of Lemke (1990) and Wagenschein (1980), we can offer the following recommendations:

1. Give students more practice talking (computer) science; encourage students to talk about CS to one another and to talk in everyday language. This way, they will get used to using the terminology presented in class.
2. Teach students how to describe phenomena and CS concepts in everyday language. Students will then be able not only to communicate with other computer scientists but also to mediate and explain to other learners.
3. Require your students to be able to say anything in more than one way and always in full sentences to make sure about the meaning. This allows the teacher to check for understanding and to see where there are still problems that need to be addressed.
4. As a teacher, prepare and provide different explanations in everyday language and in scientific language for the same concept and make it explicit when you use each, for example do I make a difference in the use of 'programming' and 'coding'? Where is this difference, and how can I explain it to my students in everyday language?
5. Teach the meta-discourse; discuss the different meanings of terms, their usage and their origins and metaphors, for example show different definitions from textbooks and negotiate on a set of central terms for that course and their meanings and write down a common glossary. A glossary can help students in class, when doing homework, to get used to terminology and also when they prepare for tests.
6. Practice with your students translating from everyday language into scientific language and back again, and grade this activity.
7. Get familiar with the everyday language that your students use, who are different from you, have different cultural backgrounds, are of a different gender, are non-native speakers and have special needs.
8. Emphasize the human side of CS, for example in modelling processes or about information technology as an everyday phenomenon and CS as one way of describing the world and so on.
9. Involve your students in decision processes on the contexts and on examples for the concepts you'd like to teach. This way, the students' connection to CS will be strengthened and your lessons will have more meaning to the students. That way, they might talk more about CS with their peer groups.
10. Give room for a discussion about the nature of computer science. Teach your students about different viewpoints on our discipline, on the different scientific methods we use and on their benefits for different application domains.

Key points

- The scientific language of computer science consists of many metaphors that might cause misunderstandings.
- Students and teachers need to talk about computer science to one another to negotiate meanings of terms and to avoid misunderstandings.
- Everyday language, educational language and scientific language have to be considered by teachers to support learning.
- When introducing a new term, students and teachers should reflect on other meanings in non-CS-contexts before using it scientifically.
- The framework of educational reconstruction helps plan lessons from different perspectives.
- The meta-discourse about CS terms and their usage supports learning and connecting CS terms and knowledge to everyday life.

For further reflection

Try using meta-discourse within a workshop with your colleagues: Think about the terms 'coding', 'implementing', 'modelling' and 'programming'. Write down on your own one typical sentence you use for each term. Let your colleagues write down a sentence for each term also. Then, compare your sentences, discuss them and agree on one definition for each term and on one exemplary sentence for each of them. Write them on a poster, and place it in your classroom or another frequented wall in your school.

Repeat this with another set of terms with 'quite the same but different' meanings.

References

Alexander, R. (2006), *Towards Dialogic Teaching: Rethinking Classroom Talk*, Cambridge: Dialogos.

Anton, M. A. (2010), 'Wie heißt das auf Chemisch? Sprachebenen der Kommunikation im und nach dem Chemieunterricht', in G. Fenkart, A. Lembens, E. Erlacher-Zeitlinger (eds), *Sprache, Mathematik und Naturwissenschaften*, Innsbruck: StudienVerlag GesmbH.

Braught, G., Eby, L. M. and Wahls, T. (2008), 'The Effects of Pair-Programming on Individual Programming Skill', in *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, 200–4.

Busch, C. (1998), *Metaphern in der Informatik—Modellbildung—Formalisierung—Anwendung*, Wiesbaden: Deutscher Universitäts-Verlag.

Cui, R., and Teo, P. (2020, October), 'Dialogic Education for Classroom Teaching: A Critical Review', *Language and Education*, 35 (3): 187–203. https://doi.org/10.1080/09500782.2020.1837859.

Cummins, J. (1979), 'Linguistic Interdependence and the Educational Development of Bilingual Children', *Review of Educational Research*, 49: 222–51.

Cummins, J. (2008), 'BICS and CALP: Empirical and Theoretical Status of the Distinction', *Encyclopedia of Language and Education*, 487–99, Berlin: Springer.

Dawes, L. (2004), 'Talk and Learning in Classroom Science', *International Journal of Science Education*, 26 (6): 677–95.

Diethelm, I., and Goschler, J. (2014), 'On Human Language and Terminology Used for Teaching and Learning CS/Informatics', in *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, New York, 122–3, ACM.

Diethelm, I., Hubwieser, P., and Klaus, R. (2012), 'Students, Teachers and Phenomena: Educational Reconstruction for Computer Science Education', in R. McCartney and M.-J. Laakso (eds), *12th Koli Calling Conference on Computing Education Research*, 164–173, Tahko: ACM.

Diethelm, I., and Zumbrägel, S. (2010), 'Wie funktioniert eigentlich das Internet? Empirische Untersuchung von Schülervorstellungen', in *Proceedings of Didaktik der Informatik. Möglichkeiten Empirischer Forschungsmethoden und Perspektiven der Fachdidaktik*, 33–44, Bonn.

Izwaini, S. (2003), 'A Corpus-Based Study of Metaphor in Information Technology', in *Proceedings of the Workshop on Corpus-Based Approaches to Figurative Language Lancaster*, 110, UK.

Lakoff, G., and Johnson, M. (2003), *Metaphors We Live By*, Chicago: University of Chicago Press.

Lemke, J. (1990), *Talking Science—Language, Learning, and Values*, Westport, CT: Ablex.

Lewis, C. M. (2011, June), 'Is Pair Programming More Effective Than Other Forms of Collaboration for Young Students?' *Computer Science Education*, 21 (2): 105–34. https://doi.org/10.1080/08993408.2011.579805.

McDowell, C., Werner, L., Bullock, H. E., and Fernald, J. (2006), 'Pair Programming Improves Student Retention, Confidence, and Program Quality', *Communications of the ACM*, 49 (8): 90–5.

Mercer, N. (1995), *The Guided Construction of Knowledge: Talk amongst Teachers and Learners*, Bristol: Multilingual Matters.

Nystrand, M., Wu, L. L., Gamoran, A., Zeiser, S., and Long, D. A. (2003), 'Questions in Time: Investigating the Structure and Dynamics of Unfolding Classroom Discourse'. *Discourse Processes*, 35 (2), 135–98.

Porter, L., Bailey Lee, C., Simon, B., and Zingaro, D. (2011), 'Peer Instruction: Do Students Really Learn from Peer Discussion in Computing?' in *Proceedings of the Seventh International Workshop on Computing Education Research*, 45–52.

Rincke, K (2011), 'It's Rather Like Learning a Language: Development of Talk and Conceptual Understanding in Mechanics Lessons', *International Journal of Science Education*, 33 (2): 229–58.

Schulte, C. (2008), 'Block Model: An Educational Model of Program Comprehension as a Tool for a Scholarly Approach to Teaching', in *Proceedings of the Fourth International Workshop on Computing Education Research (Sydney, Australia) (ICER '08),*149–60, New York: ACM.

Sentance, S., and Waite, J. (2021, August), 'Teachers' Perspectives on Talk in the Programming Classroom: Language as a Mediator', in *Proceedings of the 17th ACM Conference on International Computing Education Research*, 266–80.

Steffen, K. (2006), 'Metaphern in der Informatik'. http://waste.informatik.hu-berlin.de/diplom/staatsexamensarbeiten/steffen.pdf.

Vahrenhold, J., Cutts, Q., and Falkner, K. (2019), *Schools (K–12): In The Cambridge Handbook of Computing Education Research*, Sally A. Fincher and Anthony V. Editors Robins (eds), 547–83, Cambridge: Cambridge University Press.

Wagenschein, M (1980), *Naturphänomene sehen und verstehen*, Stuttgart: Klett.

Zakaria, Z., Vandenberg, J., Tsan, J., Boulden, D. C., Lynch, C. F., Boyer, K. E. and Wiebe, E. N. (2021), 'Two-Computer Pair Programming: Exploring a Feedback Intervention to improve Collaborative Talk in Elementary Students', *Computer Science Education*, 32 (1): 3–29.

Zingaro, D. (2014), 'Peer Instruction Contributes to Self-Efficacy in CS1', in *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14),* 373–8, New York: ACM. https://doi.org/10.1145/2538862.2538878.

15

Investigating Attitudes towards Learning Computer Science

Quintin Cutts and Peter Donaldson

Chapter outline

Chapter synopsis

This chapter explores computer science learners' attitudes towards learning, as well as those of their teachers, and how teaching practices may affect these attitudes. The exploration is set in the context of Carol Dweck's Mindset, a concept with which many readers will be familiar or will have at least read about (Dweck, 2008). Mindset concerns how learners' beliefs about their ability to learn directly affect their learning outcomes, and Mindset-based interventions have been successfully applied in computing contexts, for example our own study (Cutts et al., 2010). However, Mindset has subtle consequences for teachers, which are often lost. For, as we shall show, teachers' own Mindsets and the practices they adopt are instrumental in the formation of their learners' Mindsets. Teachers must explore their own attitudes and practices carefully and examine why their students may be getting blocked in their studies. As we move

from being an optional or self-selected subject at late secondary or tertiary education to a mainstream school subject starting at the primary level and taken *by all*, this exploration is ever more important if we are to expect success for our learners. An exercise is provided in the chapter to help the exploration, which tends to lead to a re-evaluation of our underlying understanding of how learning and skill development work, encouraging us to raise important questions about our own practices.

15.1 Introduction to Mindset

Carol Dweck's thirty-year research programme, broadly captured under the title Mindset, aims to foster more appropriate attitudes and practices among learners and teachers so that learning outcomes can be improved. It centres not on the identification of some natural ability as the predictor of learning success, but instead on the learner's attitude to whether they *think* they can get better. In summary, those who do believe they can improve have a so-called *growth* Mindset, whereas those who think ability levels are traits one is born with have a *fixed* Mindset.

It is important to note that Dweck is not disputing the observation that some people find some types of activities or learning easier than others. What she disputes is that others can't learn (Dweck, 2008). Some key aspects deriving from the Mindset work are as follows:

- A person's Mindset can vary depending on which skill or ability is considered. In everyday life, we hear this all the time, for example: 'Oh I just can't do maths at all!' or 'I haven't got an ear for music!'. In our discipline, Scott and Ghinea (2014) provide evidence that students' Mindsets relating to computer science ability are distinct from general views about intelligence.
- Mindset can be measured and is malleable. Although the word 'Mindset' might imply something *set* and unchangeable, Mindset is in fact highly malleable. Experiences that learners go through can influence their Mindset towards certain fields of study.
- Educationally, *performance* and *mastery* goals are associated with fixed and growth Mindsets (Dweck and Legget, 1988). Fixed Mindset learners tend to value performance – that is, some outcome measure of a learning exercise, whereas those with a growth Mindset value gaining mastery of the subject matter. In computing, for example, a performance measure may be a running program as the outcome of a programming exercise – for a fixed Mindset learner, as long as the program is working, all is good; a more mastery-oriented valuation of the exercise would be that the learner fully understands how the program works or that they had learned enough from the exercise so that they could solve a related problem. Zingaro and Porter (2016) showed that mastery goals evidenced in computer science students were a predictor of continued interest and success in the subject.
- Directly related to the performance and mastery goals, learners with different Mindsets adopt different *responses to feedback*. Growth Mindset learners are likely to largely ignore a mark on a piece of work (a performance measure) and concentrate on feedback that will help them to improve next time. By comparison, studies show that fixed Mindsetters focus only on the mark and do not cognitively engage with formative feedback that could help them improve; instead

they are stuck in the emotional centres of the brain (Mangels et al., 2006). In the face of failure to achieve a goal, which is in itself a kind of feedback, those with a growth Mindset will try again using one of a number of alternative strategies in order to reach the goal; fixed Mindsetters tend to either give up or hopelessly retry using the same strategy over and over. Dempsey et al. (2015) demonstrated that improving students' self-efficacy is an important goal for success in computer science, and having multiple strategies in the face of difficulties will clearly enhance self-efficacy.

Key concept: Mindset

Our *Mindset*, or our attitude towards whether we can learn something, is at least as important as any natural ability we may have. We can have different Mindsets towards different subjects to be learned. Our Mindset towards learning a subject can change from growth to fixed and vice versa. A growth Mindset person has the learning goal to master a subject; a fixed Mindset person will only attempt to perform well. Growth Mindset learners make use of feedback; fixed Mindset learners favour a rating or mark, ignoring formative feedback.

15.2 Should Mindset concern only the learners?

Much of the Mindset work appears to focus principally on the learner. In the description of Mindset in the previous section, it is the learners' beliefs that influence their ability to succeed; it is the learners' response to feedback that will keep them stuck or enable them to progress; it is the learners' willingness to try a new strategy in the face of failure that will ultimately get them to the goal. In studies, the learners' Mindsets are measured, and interventions are designed to directly influence the learners' beliefs, via, for example, the messages mentioned above.

In computer science Mindset studies, the focus has also been on the learners. In Simon et al. (2008), students took part in a so-called 'saying is believing' intervention, in line with earlier successful studies (Aronson, Fried and Good, 2002): having learned about Mindset in a short lecture segment, the students wrote about applying Mindset ideas in the context of solving difficult problems. This intervention was relatively separated from their actual computer science work, and ultimately, changes in their computer science Mindset were not significant. If anything, they became more fixed as they went through the course, a result also found by Scott and Ghinea (2014). In our own study (Cutts et al., 2010), more intensive Mindset training did produce significant changes in the students' Mindsets, but again, the focus of the training was the learners.

Some questions come to mind:

- What is the influence of *teachers*' Mindsets on the learning of those in their care?
- Does influencing the learners' Mindsets directly, via some form of instruction about Mindset concepts, lead to increased learning? What is the influence of the instructional design itself on the learners' Mindsets and consequently their learning?

15.3 Considering teachers' Mindsets

We will address the second issue raised above later in the chapter. On the first question, studies of managers were the first to show that the Mindset of those in a mentoring/training role was related to their actions towards their charges (Heslin, Vanderwalle and Latham, 2006). Where the manager had a growth Mindset, he or she was more likely to coach an employee in difficulties and was able to come up with more suggestions for going forward. Similar effects have been observed in studies of teachers.

Do computer science teachers hold a fixed Mindset towards learning the subject? Large numbers appear to do so:

- Lewis (2007) polled junior and senior students and also academic staff asking them to rate their agreement to the statement 'Nearly everyone is capable of succeeding in the computer science curriculum if they work at it.' Seventy-seven per cent of the staff disagreed with the statement. In Kinnunen et al.'s (2007) study, the majority of the sixteen CS educators interviewed talked about students on their courses who just 'get it' and about those who never will.
- The fact that there is heated on-going debate around whether computer science/programming ability is innate or learned speaks to the lack of consensus on this issue. One of this chapter's author's own twenty-year computer science education research career started with this question (Cutts, 2001); Ahadi and Lister (2013) introduce the term 'geke gene' while arguing that innate ability is only part of the picture; Guzdial (2014) argues against the finding in Macnamara, Moreau and Hambrick (2016) that deliberate practice only minimally explains expert performance; and Patitsas et al. (2016) demonstrate that bimodal distributions in computer science don't exist and are a folklore based on instructors' prejudices about innate ability. Most recently, McCartney et al. (2017) have proposed a malleable 'Geekiness Quotient', which balances the concepts of innate ability and acquired skill.
- In our own work with teachers over the years, we have come across those who undoubtedly pigeonhole their pupils into ability levels, demonstrating a fixed Mindset attitude. During the CS Inside project (Cutts et al., 2007) which was an early adopter of the *Unplugged* style of computer science education, teachers often expressed surprise when particular pupils, obviously generally viewed to be low-achieving, made more valuable contributions than their more highly rated peers during sessions using this alternative pedagogical approach.

From all of this, it therefore seems essential that we should be considering *teachers*' Mindsets at least as enthusiastically as those of the *learners*, for, from the former, directly or indirectly, are created the latter. The takeaway message here is that *we, the teachers, are actually at the heart of the Mindset issue, not the learners*.

15.4 Exploring our own Mindset and practices as teachers

Surprisingly, most teachers we've encountered tell us that they are aware of Dweck's Mindset work generally, yet their comments and actions belie a fixed Mindset attitude. Dweck herself, on the basis of numerous studies, for example Sun (2015), identifies the notion of a *false growth* Mindset, where teachers state they have a growth Mindset but don't act as though they do.

Having read this far, where do you think you stand? What is your own view of the potential for any learner to develop their computer science ability? This is a fundamentally important question as computer science is globally transformed from its erstwhile position as an optional subject taken only by those who chose it in upper secondary or tertiary education into a mandatory part of all pupils' education starting in early primary. Even as an optional subject, when one might sensibly conjecture that those taking it would have some expectation of success, computer science has had a notoriously high failure rate (Watson and Li, 2014). If one holds a view of computer science ability as innate (fixed Mindset), how can one then subject young people everywhere to a computer science education programme, knowing that large numbers must fail?

Here is a question to facilitate an exploration of our position. *Is our stated expectation of what learners can achieve* (i.e. our Mindset) *at odds with our own teaching practices or those we have experienced?* A teacher who believes that computer science ability is innate (fixed Mindset) most likely has adopted, consciously or unconsciously, a set of teaching practices that mirror his or her belief. For example, the programme may move too fast, provide little feedback, offer no opportunity for catching up and so on.

More importantly, however, even if a teacher says that he or she has a growth Mindset attitude towards the learning of computer science, we conjecture that very often the learning and teaching approaches used are still not sufficiently supportive – they are still likely to foster a fixed Mindset attitude in learners.

Do you disagree with our conjecture? Are you outraged that we might suggest this disconnection between growth Mindset belief and fixed Mindset practices? If so, then consider the following exercise, which we have used with large numbers of teachers to shed light on the disconnect.

The exercise starts with a familiar pseudocode example – making a cup of tea – as is often used as a first example of creating a plan. Don't be side-tracked though: this is nothing to do with program planning, and any example algorithm written in plain English would be fine. Take your time answering the questions and don't read ahead! So, the example is as follows:

1. Fill the kettle.
2. Flick the switch on.
3. While the kettle is boiling, get a cup and put a teabag into it.
4. Once boiled, pour water into the cup.
5. Wait a few minutes.

6. Remove the teabag.
7. Add milk to taste.

The first question we have posed to numerous teachers about this is as follows: *What do you need to know to understand these instructions in plain English as a solution to the task of making a cup of tea?* We do this at a whiteboard and ask the group to shout out the answers – take a moment yourself now to consider your own answer. You should range right from issues to do with tea-making down to elements of syntax.

This takes a while, and there's usually a lot on the board by the time we've finished. The following categories at least usually emerge from the discussion:

- Appreciating the gross structure of a set of steps within the body of text.
- Being able to parse sentences, seeing the structure and picking out verbs as actions or operations, nouns as things and all the necessary grammatical connecting tissue.
- A hierarchy of concepts, from those specific to this problem to more general concepts – from tea-making specifically, such as what teabags are, the steeping time necessary for tea to brew, to liquids more generally, such as milk, water, boiling, kettle and cups, to primitive process concepts such as sequence, parallel activity, time and pauses.
- Contextual connections, some implicit such as in line 6 – remove the teabag *from where?* – some explicit such as kettle, cup and teabag being referred to in many places, and across many lines, for example switch on/once boiled.

We then ask: *How long does it take to learn about all of this for the typical human being?* There's a bit of thinking, before answers of *around five to six years* emerge.

We then transition to a program fragment of ten to fifteen lines in a language that the teachers are familiar with, using constructs that they'd use with their pupils. It could be any simple program, but this one reads in a mark that should be out of thirty, checks it is in range and repeatedly reads a new number in if it is not and finally converts the validated mark to a percentage and writes it out:

```
Dim Mark as Integer
Dim Percentage as Single
Dim OutOfRange as Boolean
Percentage = 0
OutOfRange = True
Do
Mark = InputBox('What mark did you get? ')
OutOfRange = Mark < 0 Or Mark > 30
If OutOfRange Then
MsgBox('Error! Mark not in range 0-30')
Endif
While OutOfRange
MsgBox('You got ' & (Mark/30)*100 & '%')
```

We ask the same question again: *What do you need to know to be able to read and understand this?*

An even longer list of categories tends to emerge, including the following:

- Being able to see the gross structure of the program, including variable declarations and initializations, the loop to validate the input, the calculation and the output of the result
- Being able to remember all the syntactic elements as well as all their meanings
- Understanding that a formal language is different from a natural language (related to Pea's (1986) *superbug* and further explained in Tenenberg and Kolikant (2014) and Cutts et al. (2014))
- Variables, including what they are as a concept, what the types associated with them mean, what an assignment is
- Expressions, including literal values in the program text such as True, 0, 30, 'You got', simple binary operations and then complex expressions combining a number of operators, either numeric or logical as well as concatenation and type coercion
- What input and output library functions do, and how to use them
- Control flow constructs of sequence and conditional loops
- Understanding the context – exam marks and how they can be out of range and arithmetic conversion to percentages

When asked the related question *How much time do you give your pupils before expecting them to be able to understand a program like this?*, the typical answer is *about three weeks*.

The comparison of six years to three weeks is obviously not entirely fair, but unpicking in this way how much there is to be learned, deeply, by comparing natural and programming language texts is often a radical eye-opener for teachers. In any introductory course, these concepts would be expected to be understood in three to five weeks, and the introduction of new concepts will build on these. Studies and learning models now indicate that missing the early building blocks is a key signal for failure in a course, including Robins' Learning Edge Momentum (2010), Ahadi and Lister's stumbling points (2013), and Porter, Zingaro and Lister's prediction of student success on the basis of analysing fine-grained clicker data (2014).

In the next section we use the literature to strengthen the argument that the typical instructional design in introductory computer science courses is likely also to foster a fixed Mindset in learners.

15.5 Do typical computer science learning designs foster a fixed Mindset?

When reviewing the available evidence from both the Mindset-specific studies and more general research that examines how students' attitudes and beliefs change over the course of their CS education, there is evidence that our courses are fostering a fixed Mindset. For example, Scott and Ghinea (2014) found that nearly a third (30.2 per cent) of students' beliefs about programming aptitude became more fixed, with nearly a fifth (18 per cent) of these changing completely from a growth Mindset to a fixed Mindset. Even in cases where an introductory course has been specifically designed to provide extra support, there is evidence of a modest reduction in growth Mindset. For example, in our own Mindset study (Cutts et al., 2010), those who did not receive Mindset training experienced a 5 per cent reduction in growth Mindset score over just a short six-week period.

More broadly, studies like those of Settle, Lalor and Steinbach (2015) indicate decreases in student confidence and engagement during their initial CS1 experience.

To begin to understand the reasons why, let us consider the typical instructional design of an introductory computer science course. This involves covering a range of programming concepts, one at a time; each concept is introduced using example code fragments and the output they produce, leaving learners to individually come up with an internalized model for how the fragment actually works, a small number of worked examples may be explored and learners are then given programming problems to work on, where they are expected to develop working programs using a development environment. This is a generally accepted format whether using an educational programming environment such as Scratch or an industry language such as Java.

Examining this design against the key Mindset aspects introduced at the start of the chapter, we can explore how the fixed Mindset may be being developed:

- *Performance versus mastery goals*: The predominant measure of success in a programming course is whether the learner is able to get a working solution to a problem. The running program is taken as a proxy for successful learning, but in truth, how much do we know about what the learner has in fact learned? Has the learner been asked to explain the program in detail or the process followed to get the solution? Usually not. Do we evaluate how much help was received from tutors, peers, the internet and so on to get the program working? Usually not. The dominant concept of running a program is further underlined by the increasing use of automated acceptance tests for marking and recording progress. It doesn't matter how the student gets the program, as long as it passes the test. This whole approach will clearly foster a performance goal in learners – learning to program is presented as an exercise in getting programs to run, not in understanding why they do or don't work. This is backed up by Buffardi and Edwards (2015), who found that the automatic grading system was discouraging reflective testing, an indicator of deep thinking about programs.
- *Response to feedback*: What kind of feedback do learners typically receive? In a typical context where learners attempt to solve problems by getting a working program, the programming environment provides feedback every time the learner tries to run their program. The ratio of failure to success is high in favour of failures, which in itself is disheartening to someone not used to it. Furthermore, the error messages of the programming environment are usually quite opaque, and many learners take no useful information from them that might help them identify the cause of the problem. Given these factors, it is hardly surprising that the typical learner response to all this negative feedback is to put their hand up and ask for help or to get help from peers or the web. Students with this response were characterized by Perkins et al. (1986) as *Stoppers*.
- *Alternative strategies*: We rarely explicitly teach debugging strategies, which would provide learners with options when faced with challenges. Anecdotally, more experienced students have often commented on how essential the web is as a tool to be used to find the cause of problems they're facing, but teaching students about appropriate search techniques for

solving problems is not common practice. For students who are struggling, the predominant behaviour observed by tutors and teachers is of students making almost random changes to their program, over and over again, in the hope that it will magically start working. This is a clear sign of the adoption of a fixed Mindset and the repeated application of a hopeless strategy that is not working. Such students were characterized by Perkins et al. (1986) as *extreme movers*.

You may well ask at this point why *all* learners don't fail, given the apparent weakness in typical learning designs that we are describing. Furthermore, given that some succeed, it is easy to attribute their success to some innate ability, the exact explanation the general Mindset philosophy is arguing against! In the next section, therefore, we explore an alternative explanation for why some succeed while others fail.

15.6 The influence of prior experience on novices' success

At the core of this explanation are the *prior experiences* that contribute to the learners' performance. Those who have succeeded at some form of programming before are very likely to succeed in an introductory course (Ramalingam, LaBelle and Wiedenbeck, 2004). But how do we explain those who have never programmed before? Considering aspects of the computer science concepts outlined in the example above that a novice must master, the following can be concluded:

- Experience with languages of a more formal nature, including mathematical algebras and anecdotally, Latin and Greek, will have given the learner skills to appreciate both the nature of a formal language as one with strict definitions and an experience of reading texts out of order, for example precedence rules in maths, verbs at the end of the sentence in Latin and German and so on. The experience of working with mathematical language and processes may explain why studies have shown a correlation between prior high school maths performance and success in computer science course (Bennedsen and Caspersen, 2005; Bergin and Reilly, 2005).
- It is well known that a single construct, such as variable assignment, may have multiple valid interpretations (Bornat, Dehnadi and Barton, 2012). Teaching styles that introduce concepts by example enable the learner to adopt any one of these so-called alternative conceptions (Sorva, 2013), which will work for some exercises they complete, but not all (Ma et al., 2011).
- Studies show that those with visualization skills tend to achieve better in STEM subjects, including computer science (Cooper et al., 2015; Sorby, 2013). More importantly, the same studies have showed that the visualization can be acquired with training.

In summary, there is sufficient evidence to propose that the spread of outcomes in our classes is at least partly based on prior experiences.

15.7 The bigger picture: Teacher and learner attitudes and learning designs

We have now laid sufficient groundwork to come back to the questions posed but not answered earlier in the chapter:

- Does influencing the learners' Mindsets directly, via some form of instruction about Mindset concepts, lead to increased learning?
- What is the influence of the instructional design itself on the learners' Mindsets and consequently their learning?

With respect to the second question, we have shown that typical instructional designs are highly likely to foster a fixed Mindset towards computer science learning in learners with limited relevant prior experiences. Hence, we may now expect the answer to the first question to be no. To back up this expectation, the minimal Mindset training intervention of Simon et al. (2008) didn't influence the students' Mindsets, let alone increase their learning. The more in-depth Mindset training of our study (Cutts et al., 2010) did influence students towards more growth Mindsets, but, crucially, the Mindset training alone was not enough to increase learning. In the study, three interventions were used: one was the Mindset training, a second explicitly encouraged the students to try new strategies in the face of failure and the third emphasized the importance of making good use of available high quality feedback to their learning. Only when students had both the Mindset training and one of the other interventions were improvements in learning noted.

The importance of this result is that the additional growth Mindset-based interventions changed the instructional design of the course and were the result of the teacher fully taking on board the ramifications of Mindset as laid out in Dweck's work and the many related studies.

As Dweck said, teaching with a growth Mindset is not a statement, it's a journey. The journey requires a deep re-evaluation of our own understanding of how learning computer science skills can take place, from an understanding of both general education theory and the increasing understanding we have about computer science education in particular. It requires a continual investigation of our own practices and how they affect students' success. It must inevitably lead to a major rethink of the typical instructional design for an introductory computer science course.

Key points

- Mindset is the belief that people hold about whether ability is mostly innate (fixed) or can be improved through deliberate practice in a specific area of learning (growth).
- Educationally, performance and mastery goals are associated with fixed and growth Mindsets, respectively – Mindset has a huge influence on outcomes.

- Mindset towards a particular skill or ability can be measured and can, and does, change with learners able to move between growth and fixed Mindsets depending on their learning experience.
- A teacher's Mindset is easily transferred to their learners. It's crucial therefore that teachers develop a growth Mindset in order to create instructional designs that can foster a growth Mindset in learners.
- Teachers of computer science have been shown to have a fixed Mindset. Exercises such as the one in this chapter help to shift them towards a growth Mindset.
- Typical learning and teaching techniques in computer science foster a fixed Mindset.
- Just saying 'I believe in growth Mindsets' isn't enough – a teacher needs to re-evaluate their teaching practices to determine how they could be influencing learners' Mindsets.

For further reflection

Examining the design of your own course or programme of learning,

- do you get learners to carry out a survey or exercise at the start of the course to help you judge the beliefs they start with and again towards the end of the course to see how they've changed?
- have you provided a range of suitable strategies that learners can use when they get stuck on known areas of difficulty?
- can you reorganize tasks so that learners have an opportunity to directly apply the feedback they receive soon afterwards?
- do you have mechanisms in place for learners to move at different speeds?
- for programming in particular, do you explain that repeated failure is to be expected? – programs don't work correctly for most of the development period and then finally do!

References

Ahadi, A., and Lister, R. (2013), 'Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant?', Paper presented at the Ninth ACM International Computing Education Research Conference (ICER 13), San Diego, CA.

Aronson, J., Fried, C. B., and Good, C. (2002), 'Reducing Stereotype Threat and Boosting Academic Achievement of African-American Students: The Role of Conceptions of Intelligence', *Journal of Experimental Social Psychology*, 38: 113–25.

Bennedsen, J., and Caspersen, M. E. (2005), 'An Investigation of Potential Success Factors for an Introductory Model-Driven Programming Course', Paper presented at the First ACM International Computing Education Research Conference (ICER '05), Seattle, WA.

Bergin, S., and Reilly, R. (2005), 'Programming: Factors That Influence Success'. Paper presented at the Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education, St. Louis, MI.

Bornat, R., Dehnadi, S., and Barton, D. (2012), 'Observing Mental Models in Novice Programmers'. Paper presented at the 24th Annual Workshop of the Psychology of Programming Interest Group, London, UK. http://www.ppig.org/library/paper/observing-mental-models-novice-programmers.

Buffardi, K., and Edwards, S. (2015), 'Reconsidering Automated Feedback: A Test-Driven Approach'. Paper presented at the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15), Kansas City, KA.

Cooper, S., Wang, K., Israni, M., and Sorby, S. (2015), 'Spatial Skills Training in Introductory Computing'. Paper presented at the Eleventh ACM International Computing Education Research Conference (ICER '15), Omaha, NA.

Cutts, Q. (2001), 'Engaging a Large First Year Class', in M. Walker (ed.), *Reconstructing Professionalism in University Teaching*, 105–28, Buckingham: Open University Press.

Cutts, Q. I., Brown, M. I., Kemp, L., and Matheson, C. (2007), 'Enthusing and Informing Potential Computer Science Students and Their Teachers', *ACM SIGCSE Bulletin*, 39 (3): 196–200.

Cutts, Q. I., Connor, R. C. H., Donaldson, P., and Michaelson, G. (2014), 'Code or (not Code) – Separating Formal and Natural Language in CS Education'. Paper presented at the Workshop in Primary and Secondary Computing Education (WiPSCE '14), Berlin, Germany.

Cutts, Q. I., Cutts, E., Draper, S., O'Donnell, P., and Saffrey, P. (2010), 'Manipulating Mindset to Positively Influence Introductory Programming Performance', Paper presented at the 41st ACM Technical Symposium on Computer Science Education, Milwaukee, WI.

Dempsey, J., Snodgrass, R. T., Kishi, I., and Titcomb, A. (2015), 'The Emerging Role of Self-Perception in Student Intentions', Paper presented at the Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15).

Dweck, C. (2008), *Mindset: The New Psychology of Success*, New York: Ballantine.

Dweck, C. S., and Legget, E. L. (1988), 'A Social-Cognitive Approach to Motivation and Personality', *Psychological Review*, 95 (2): 256–73.

Guzdial, M. (2014), The 10K Hour Rule: Deliberate Practice Leads to Expertise, and Teaching Can Trump Genetics. https://computinged.wordpress.com/2014/10/13/the-10k-rule-stands-deliberate-practice-leads-to-expertise-and-teaching-can-trump-genetics/.

Heslin, P., Vanderwalle, D., and Latham, G. (2006), Keen to Help? Managers' Implicit Person Theories and Their Subsequent Employee Coaching', *Personnel Psychology*, 58: 871–902.

Kinnunen, P., McCartney, R., Murphy, L., and Thomas, L. (2007), 'Through the Eyes of Instructors: A Phenomenographic Investigation of Student Success', Paper presented at the Third ACM International Computing Education Research Conference (ICER '07), Atlanta, GA.

Lewis, C. (2007), 'Attitudes and Beliefs about Computer Science among Students and Faculty', *SIGCSE Bulletin*, 39 (2): 37–41.

Ma, L., Ferguson, J., Roper, M., and Wood, M. (2011), 'Investigating and Improving the Models of Programming Concepts Held by Novice Programmers', *Computer Science Education*, 21 (1): 57–80.

Macnamara, B., Moreau, D., and Hambrick, D. (2016), 'The Relationship between Deliberate Practice and Performance in Sports: A Meta-Analysis', *Perspectives on Psychological Science*, 11 (3): 333–50.

Mangels, J. A., Butterfield, B., Lamb, J., Good, C. and Dweck, C. S. (2006), 'Why Do Beliefs about Intelligence Influence Learning Success? A Social Cognitive Neuroscience Model', *Social Cognitive and Affective Neuroscience*, 1 (2): 75–86.

McCartney, R., Boustedt, J., Eckerdal, A., Sanders, K. and Zander, C. (2017), 'Folk Pedagogy and the Geek Gene: Geekiness Quotient', Paper presented at the 48th ACM Technical Symposium on Computing Science Education (SIGCSE '17), Seattle, WA.

Patitsas, E., Berlin, J., Craig, M., and Easterbrook, S. (2016), 'Evidence That Computer Science Grades Are not Bimodal'. Paper presented at the Twelfth ACM International Computing Education Research Conference (ICER 16), Melbourne, Australia.

Pea, R. D. (1986), 'Language-Independent Conceptual "Bugs" in Novice Programming', *Educational Computing Research*, 2 (1): 25–36.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., and Simmons, R. (1986), 'Conditions of Learning in Novice Programmers', *Journal of Educational Computing Research*, 2 (1): 37–55.

Porter, L., Zingaro, D., and Lister, R. (2014), 'Predicting Student Success Using Fine Grain Clicker Data'. Paper presented at the Tenth ACM International Computing Education Research Conference (ICER '14), Glasgow, Scotland.

Ramalingam, V., LaBelle, D., and Wiedenbeck, S. (2004), 'Self-Efficacy and Mental Models in Learning to Program', Paper presented at the Ninth ACM Innovation and Technology in Computer Science Education Conference (ITiCSE '04), Leeds, UK.

Robins, A. (2010), 'Learning Edge Momentum: A New Account of Outcomes in CS1', *Computer Science Education*, 20 (1): 37–71.

Scott, M. J., and Ghinea, G. (2014), 'On the Domain-Specificity of Mindsets: The Relationship between Aptitude Beliefs and Programming Practice', *IEEE Transactions on Education*, 57 (3): 169–74. doi:10.1109/te.2013.2288700.

Settle, A., Lalor, J., and Steinbach, T. (2015), 'Reconsidering the Impact of CS1 on Novice Attitudes', Paper presented at the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15), Kansas City, KA.

Simon, B., Hanks, B., Murphy, L., Fitzgerald, S., McCauley, R., Thomas, L., and Zander, C. (2008), 'Saying Isn't Necessarily Believing: Influencing Self-Theories in Computing'. Paper presented at the Fourth International Workshop on Computing Education Research ICER '08, Sydney, Australia.

Sorby, S. (2013), 'The Case for Spatial Skills Instruction in Computer Science', *Future Directions in Computing Education Summit White Papers (SC1186)*. https://stacks.stanford.edu/file/druid:mn485tg1952/SorbySherylOhioState.pdf.

Sorva, J. (2013), 'Notional Machines and Introductory Programming Education', *ACM Transactions on Computing Education*, 13 (2): 1–64, Article 8.

Sun, K. L. (2015), 'There's No Limit: Mathematics Teaching for a Growth Mindset', PhD thesis, Stanford University.

Tenenberg, J., and Kolikant, Y. B.-D. (2014), 'Computer Programs, Dialogicality, and Intentionality'. Paper presented at the Tenth ACM International Computing Education Research Conference (ICER '14), Glasgow, Scotland.

Watson, C., and Li, F. W. B. (2014), 'Failure Rates in Introductory Programming Revisited'. Paper presented at the 2014 Conference on Innovation and Technology in Computer Science Education, Uppsala, Sweden.

Zingaro, D., and Porter, L. (2016), 'Impact of Student Achievement Goals on CS1 Outcomes'. Paper presented at the Proceedings of the 47th ACM Technical Symposium on Computing Science Education SIGCSE '16, Memphis, Tennessee.

16

Formative Assessment in the Computing Classroom

Sue Sentance, Shuchi Grover and Maria Kallia

Chapter outline

Chapter synopsis

This chapter gives a general introduction to formative assessment and applies this to the teaching of computing in school, with reference to research studies in this area. A variety of methods that can be used for assessment are discussed including self- and peer assessment, automated tools, rubrics, concept maps and multiple-choice questions. Ways of assessing programming projects are discussed and evaluated. Finally, an example of an assessment platform and the types of features that might be useful for such platforms are discussed.

16.1 Assessment overview

Assessment is critically important to education – for evaluation and accreditation and for supporting learning, with summative assessments supporting the former and formative

assessments the latter. Key aspects of formative assessment are questioning, dialogue and developing students' abilities to self-assess (Black and Wiliam, 1998), and in order to learn more, students need feedback on their progress. Students can use teacher feedback to work on their areas for development, which puts the responsibility and power into the hands of the student. Students should be able to use self-assessment to identify their areas for improvement and apply themselves to these.

In this chapter, we focus primarily on formative assessment, or assessment for learning, which includes feedback, questioning, self-assessment opportunities, peer assessment and a range of other strategies in the context of computing. While assessment opportunities such as projects or tests may be used formatively or summatively, primarily in this chapter we discuss these in terms of their contribution to formative assessment, drawing on existing research in the field.

Key concept: Formative and summative assessment

A key feature of assessment is that it concerns making judgements. This enables an assessment of learning to be made, known as summative assessment. With this type of assessment, students are graded at the end of a course or module. This can be for certification purposes or for evaluating learning in the aggregate and how well the learning goals were addressed by the curriculum.

Another type of assessment is assessment for learning, known as formative assessment, which is used to identify what students know, what they still need to learn and how to get there. This can also be used for diagnostic purposes and to address student misconceptions. There are many methods that can be used to formatively assess students, some of which are discussed in this chapter.

Formative assessment taking place at school has the intention of promoting further learning for the student, whereas summative assessment revolves around key points where a student's achievement is measured, giving data that can be used not only by the student but also by a number of other stakeholders, including the school and employers.

It is necessary to make a distinction between the assessment opportunity and the assessment activity; in this chapter we discuss both. The assessment opportunity is a task that may be set with the intention of using it for assessment. The assessment activity is the provision of feedback, dialogue and student discussion, generated as a result of the task, that provides an opportunity for improving student learning. For some tasks, feedback is inherent, but in others, it happens later. Thus, it is possible to provide an assessment opportunity that does not result in much assessment activity.

The framework shown in Figure 16.1 highlights three key aspects or dimensions of formative assessment in school – the design of assessments, teacher preparation and formative assessment literacy, and community participation and resource repositories.

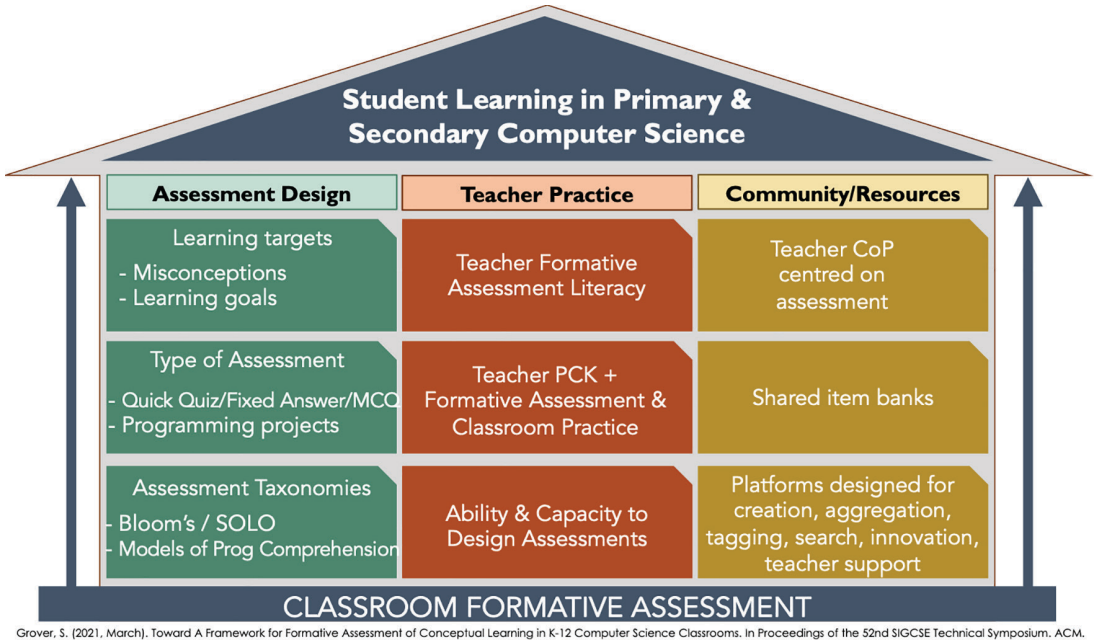

Figure 16.1 A framework for assessment (*Source:* Grover 2021)

16.2 Assessment methods

In this section we look at some of the different methods we can use to assess computer science knowledge and skills:

- Self- and peer assessment
- Parson's puzzles
- Rubrics
- Concept maps
- Response systems and multiple-choice questions (MCQs)

Self- and peer assessment

Peer and self-assessment are effective methods of formative assessment, both useful in the classroom. Boud and Falchikov (2007) define peer assessment as a process of providing feedback on peers' work based on success criteria that the students may have established previously. Self-assessment can be defined as formative assessment evaluating the extent to which oneself or one's work meets requirements or success criteria. Peer review or assessment has been used as a learning process to improve the quality of computer programs for at least thirty years (Luxton-Reilly, 2009).

Example: Using peer assessment in the classroom

Ask students to write an algorithm to control a red, amber and green traffic light. Writing the algorithm is the assessment opportunity. Using peer assessment, the students can then share their algorithms, either on paper or by presenting them in small groups. You should decide on some criteria to assess the algorithms such as 'functionality', 'notation' and 'improvements'. The writing and giving of feedback is the assessment activity. At the end of the peer assessment, students should understand their strengths and where they need to improve, possibly with recommendations on how to do so. The more often you carry out peer assessment, the more familiar students get, and they will be able to give increasingly constructive feedback to their peers as well as having a growing understanding of what the criteria mean.

By engaging with peer assessment, students move from being merely observers in the classroom to being more active participants, engaging with the learning process. In addition, the levels of collaboration and sense of community are increased by the use of peer assessment (Clark, 2004). Topping (2009) suggests that peer assessment helps students really understand the aims and objectives of a course. This leads to students developing a better understanding of the assessment criteria against which they themselves will be judged. Students, assessing the work of others, benefit by developing skills to identify errors in their own work (Sitthiworachart and Joy, 2003). Self-assessment can also help students to identify errors and to further develop the ability to accurately measure their performance against requirements, which is important for creating effective learners (Boud, Lawson and Thompson, 2013).

A small-scale study, examining students' ability to self-assess their programming skills that used a survey aligned to Bloom's taxonomy, found that students' self-assessment was accurate in relation to their summative assessment (exam) scores (Alaoutinen and Smolander, 2010). In a high school environment, teachers' and students' assessment scores were highly correlated (Tseng and Tsai, 2007). However, other studies show that students demonstrate a tendency to be more generous with self-assessment marks (Sajjadi et al., 2016) and with peer assessment of their friends (Sitthiworachart and Joy, 2003). Weaker students appear less accurate when self-assessing (Murphy and Tenenberg, 2005).

Overall, the benefits of self- and peer assessment include increasing levels of engagement and collaboration in the classroom, increasing understanding of assessment criteria and identification of strengths and weaknesses in a student's work. As assessment activities, both self- and peer assessment are suitable for the computer science classroom. The value of these may be increased by the additional support from teachers in the form of graphical aids, templates and rubrics. Grover, Sedgwick and Powers (2020) share several examples of such guides that can be provided to students. Figure 16.2 is an example of a peer feedback form (that also integrates the programming project with English language arts).

Scene 1:

- ❑ It is clear what my partner's book is.
- ❑ There are two main sprites (characters).
- ❑ The backdrop matches the setting.
- ❑ There is dialogue between the characters.
- ❑ The code runs smoothly (no bugs).

Scene 2:

- ❑ The backdrop changes in Scene 2.
- ❑ The sprites (characters) continue their dialogue.
- ❑ The program switches backdrops and sprite costumes without bugs.

Evaluation

- ❑ The sequence of events makes sense.
- ❑ There are no bugs when I run the program.
- ❑ Hide/show blocks are used for sprites that don't belong in the scenes.
- ❑ The code is "cleaned up"— there are no unused blocks.

Written Feedback: *Then you can add your feedback on their project page!

 Two Stars 

Wish 

Figure 16.2 Sample peer feedback guide for a Scratch project (*Source:* Grover, Sedgwick and Powers, 2020)

Parson's puzzles

Parson's puzzles are drag-and-drop tasks which present fragments of code that students should put in the correct order to solve a problem. Parson's puzzles are supported by tools like Hot Potatoes, Ville and JSParson (Ihantola and Karavirta, 2011). Table 16.1 presents an example in the Ville environment. Students regard this type of task as easier than writing code and more creative and objective than other assessment tasks (Denny, Luxton-Reilly and Simon, 2008). The tools also provide feedback to students. Teachers can use these puzzles as an alternative to open-ended code writing questions and identify students' difficulties and misconceptions.

Rubrics

Rubrics provide a way of offering more consistent assessment of student performance. They are used to define detailed criteria that describe what students should achieve (Becker, 2003), and they

Table 16.1 Example of Parson's puzzles adapted from Ville example

Program	Exercise Description
def main(): a = 1 b = -1 a = b tmp = a b = tmp	Order the lines of code so that the function swaps the contents of variables a and b

Source: Ihantola and Karavirta (2011).

Table 16.2 An example of a rubric for a solution in a programming task

	Weak 0–3	Emerging 4–6	Good 7–8	Excellent 9–10
Solution for 1 b	An incomplete solution is implemented. It does not compile and/or run.	Runs but has logical errors. *Apply emerging if program does not use 2D array or has incorrect results*	A complete solution is tested and runs but does not meet all the specifications and/or work for all test data. *Apply good if program misses one data entry line.*	A completed solution runs without errors. It meets all the specifications and works for all test data.

Source: Adapted from Eugene, Stringfellow and Halverson (2016).

have two shared characteristics. The first is a list of criteria that are necessary for the particular task and the second is the degree of quality (Andrade, 2000). These features make rubrics proper tools to reliably evaluate students' performance, but they can help students understand their progress as well (Black and Wiliam, 2009). Popham (1997) refers to rubrics as instructional illuminators.

Rubrics have been used to assess computer programming assignments from the early 1980s (Miller and Peterson, 1980; Hamm et al., 1983) to the current day (Becker, 2003; Barney et al., 2012; Mustapha et al., 2016). An example of a rubric is shown in Table 16.2.

Rubrics are regarded as fair and reliable tools with which students can better understand what is expected of them (Barney et al., 2012) and the amount of effort needed to reach a specific level of performance (Mustapha et al., 2016). Additionally, rubrics are useful in peer and self-assessment and can help students to evaluate their performance or that of their peers. Alternatively, students can be involved in developing the rubrics and thus feel more engaged with the whole process.

Concept maps

A concept map is a schematic representation of concepts and the relationships they form and can be used by students to represent their knowledge of a subject. Figure 16.3 shows an example of a concept map.

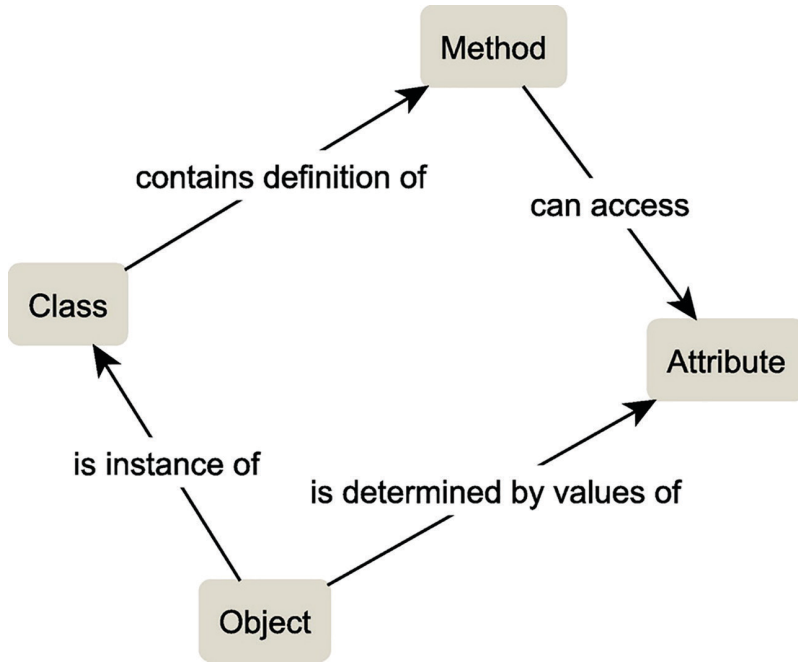

Figure 16.3 Example of a concept map (Mühling, 2016)

Table 16.3 Concept map scoring system

Component	Description	Score
Concepts	No. of concepts	1 point for each concept
Concept relationships	No. of valid relationships	1 point for each valid relationship
Branching	Scores for branching varied according to the amount of elaboration	1 point for each branching 3 points for each successive branching
Hierarchies	No. of hierarchies	5 points for each level of hierarchy
Cross-links	No. of cross-links	10 points for each cross-link
Examples	No. of examples	1 point for each example

Source: Markham et al. (1994).

As an assessment tool, it can be used to measure students' quality of understanding and the structure of their knowledge (Borda et al., 2009). Markham and Mintzes (1994) suggested a detailed scoring system (Table 16.3) to help teachers evaluate students' concept maps.

By employing concept maps, teachers can monitor students' level of learning and organization of knowledge, which leads to teachers identifying students' misconceptions and misunderstandings (Moen, 2009; Wei and Yue, 2016). In a study conducted at the university level, students reported that concept maps are more entertaining than other assessment methods (Freeman and Urbaczewski, 2001).

Drawing concept maps can also be used as a learning activity (see Chapter 13).

Response systems and multiple-choice questions

Audience response systems, informally known as 'clickers', allow the teacher to display an MCQ with optional responses. Learners respond to the question by pressing a button on an electronic device. A receiver picks up the signals from the clickers and sends the responses to software, and the software interprets and displays the results. The teacher can see the results and can choose to share them with the class.

Online versions of the 'clicker' provide more accessible use of technology in the classroom. These online systems include Kahoot, Socrative, Poll Everywhere, Edfinity and Diagnostic Questions. The facilities offered by each are slightly different, but they all incorporate a mechanism for hosting a live interactive quiz made up of MCQs. These particular systems require each student to have a device connected to the internet. Most of these are auto-gradable and therefore are very helpful in providing timely and quick feedback. Newer systems such as Edfinity.com have gone beyond MCQ types to add more interactive assessment item types such as Parson's puzzles, Point-and-Click, Hotspot interaction and matching column (Grover et al., 2022).

The use of response systems has several advantages for learners. In the classroom, a natural competitiveness among learners exists to answer the questions quickly and accurately (Cutri et al., 2016). Learners' self-confidence improves because of the opportunities for interaction with their peers (Coca and Slisko, 2013; Fotaris et al., 2016). There is some support for suggesting that the use of response systems leads to improved academic performance (Kay and LeSage, 2009), but this is contradicted by a more recent study (Wang, Zhu and Sætre, 2016). Therefore, while improvement in academic performance may not be guaranteed by the use of response systems, students perceive the systems as a positive addition to the classroom.

Online systems have advantages for the teacher. Not only do students get immediate feedback about the correct response, but the teacher gets immediate feedback about what the students know and do not know, thereby making it ideal for formative assessment (Kay and LeSage, 2009). Collecting and analysing data about individuals or groups of students is possible with the online systems. Some provide analytics that can help teachers identify gaps in knowledge by groups and individuals and feature the ability to allow comparisons to all other users of the system.

The use of response systems and online quizzes is not without disadvantages, however. Although students view the systems as engaging and motivating, an abundance of enthusiasm could lead to challenging classroom management situations. In schools, the challenge lies in acquiring the skills to use the tool as part of the learning process, rather than just as an engaging and motivating game. Students may need to be trained in how to use the online quizzes as part of the learning process and not view them just as a game.

Regardless of the delivery method, developing the MCQs is a difficult challenge (Dell and Wantuch, 2017; Kay and LeSage, 2009). Writing MCQs may seem an easy task, but writing questions to allow identification of real confusion or misconceptions is challenging (Dell and Wantuch, 2017). Some systems such as Edfinity have leaned on prior research to aggregate common misconceptions related to introductory programming concepts such as variables, functions, loops, conditionals and Boolean operators (among others) that have been documented over decades of research and create assessment items aligned to them. These items are typically

simple MCQ directly probing understanding related to the misconception and can be used as a formative assessment for timely and speedy feedback to teachers and students (Grover and Twarek, 2021).

16.3 Assessing programming projects

The challenge of assessing a programming skill is present at all levels of education, from primary to university. At university, large cohorts and the necessity to reduce marking time have led to many innovative approaches to machine marking of computer programs. However, at primary and secondary levels, the programming tasks are not as complex and the number of students at most institutions is not as large. We can consider four potential solutions to the problem of assessing programming projects:

- Assess the process
- Assess the product/artefact
- Assess by interview
- Assess the design

Assess the process

Computer science is not the first subject challenged with assessing a process as opposed to an artefact. In Israel, design-based learning (Doppelt, 2009) was an attempt to assess the thinking processes of high-school students in a course combining mechanics, electrical engineering and programming. Students documented both the process of producing their group projects and the thinking skills employed during the process. The approach of documenting the creation process, incorporating reflection and evaluation, is seen in some computing classrooms, by the use of learning journals, portfolios or coursework submissions.

Observation is another approach to assessing process. In the science laboratory, students can be observed as they engage in practical activities (Hofstein and Lunetta, 2004). An observer summatively assesses students according to planning, design, performance and analysis. This same approach may be used in the computing classroom (Brennan and Resnick, 2012).

Assess the product/artefact

The most obvious method of assessing skills in programming is to assess the artefact, including the program code created (Insa and Silva, 2015; Lobb and Harlow, 2016).

Werner, Campe and Denner (2012) analysed middle school children's games developed in the Alice visual programming language; students were able to understand and use a range of CS concepts, with about one-third using variables, one-half using built-in functions and only a single student creating a custom function.

By using Stagecast Creator, Denner Werne and Ortiz, 2012) assessed middle school girls' artefacts; on inspection of the artefacts, 18 per cent did not include interaction, only 1 per cent included an 'and if'-type test and very few changed rule names from the default of 'untitled'.

In the Scratch environment, Seiter (2015) employed the SOLO taxonomy to assess fourth-grade students' program responses; the majority of students gave relational level (one from the top) rather than extended abstract (top) responses in their code. Brennan and Resnick (2012) also assessed computational thinking through examination of the Scratch artefacts uploaded to the Scratch community website. The scripts were analysed to determine the number and distribution of the different types of blocks. They concluded that knowing the type and count of blocks used was not sufficient to evidence computational thinking.

In some way, all of these studies were uncontrolled. Some (Denner, Werner and Ortiz, 2012) lack specifications for the artefacts that may allow students to evidence their learning. Others (Brennan and Resnick, 2012; Seiter, 2015; Werner et al., 2012) provide no information about the process of producing the artefacts. Based on these studies, it seems unlikely that students will spontaneously produce evidence of learning in their artefacts.

Assess by interview

Adding another assessment instrument may strengthen the artefact-based judgements. Several researchers (Barron et al., 2002; Brennan and Resnick, 2012; Grover, Pea and Cooper, 2015; Portelance and Bers, 2015) have used interviews to fulfil this purpose.

Brennan and Resnick (2012) performed 60- to 120-minute interviews with students on the production of their Scratch projects. The responses to these questions allowed the researchers to form detailed descriptions of the development process (Brennan and Resnick, 2012). Another study conducted by Portelance and Bers (2015) employed artefact-based interviews with a group of second graders in the United States who used ScratchJr. This study is innovative because it uses peer interviewers and video interviews to assess children's learning about computational thinking.

Grover, Pea and Cooper (2015) found in their artefact-based interviews that students who did not perform well on the (mostly MCQ) summative test were able to demonstrate understanding of concepts during the interviews. This led to the conclusion to use 'systems of assessments' to get a holistic picture of student understanding.

Assess the design

One approach that forgoes employing a student-created artefact is described as design scenarios (Brennan and Resnick, 2012). In this approach, projects are created by an educator, with different projects designed that increase in difficulty and aesthetically appeal to different groups. The student can then be asked to explain what the project does, describe an extension to the project, fix a bug in the project and remix the project to add a feature (Brennan and Resnick, 2012). This process can be observed in real time, which allows understanding to be tracked. This approach still requires a

significant amount of the assessor's time. It also has the disadvantage that the programming stages of fixing a bug and remixing may be subject to unacknowledged influences, in much the same way as the artefact-only approach.

In conclusion, none of the approaches to assess programming skills and computational thinking discussed above is without disadvantages. However, there does seem to be an acknowledgement that the very popular approach of assessing code artefacts alone has shortcomings. This suggests that varied forms of assessments, or 'systems of assessments', should be employed over the course of a school year to get a multifaceted and holistic picture of student understanding (Grover, 2017). The addition of another assessment format can be used to validate the artefact assessment. This additional method could be an interview, an observation, or a test.

16.4 Edfinity: An example of an assessment platform

In order to support formative assessment practice by teachers, assessment platforms and homework systems must be feature rich to support aggregation, creation, curation and cataloguing or taxonomizing of assessments based on multiple and multilevel taxonomies relevant to CS teachers. Ongoing efforts for CS assessment item banks and repositories include Edfinity (edfinity.com), Project Quantum (https://diagnosticquestions.com/Quantum), Viva (Giordano et al., 2015) and the Canterbury Question Bank focused on introductory college-level CS (Sanders et al., 2013).

Edfinity is one example of a platform from the United States. Taxonomies on Edfinity include CS/CT topics, learning standards (such as those from CSTA, n.d.) or learning goals by curricula (such as AP CS Principles), grade, difficulty level and ad hoc metadata to support easy search and discovery. Edfinity also aids with assessment delivery, administration, auto-grading and teacher dashboards. Backend data and analytics on student performance provide teachers crucial insights into students' learning and understanding at individual and aggregate levels (Grover, Pea and Cooper, 2014).

Edfinity also provides for multiple attempts of a question, hints and feedback (or explanation) for correct and incorrect options. Solution explanations accompany the item and serve as (a form of) feedback. These explanations, as also the question stem, support rich text, graphics and video for better learner engagement and multiple modes (and languages) of presentation to equitably support diverse learners. Edfinity item types include technology-enhanced assessments that push the boundaries to include interactivity (such as hotspot and point-and-click items), drag-drop for item types such as Parson's problems, matching column and code labelling based on research in subgoal labelling (e.g. Morrison, Margulieux and Guzdial, 2015) and in-browser code entry (assessed through unit tests). Such items are not only engaging but can also help reduce cognitive load (Figure 16.4).

Assessment creation and aggregation functions on technology platforms such as Edfinity can also support features for teacher collaboration, contribution, attribution and sharing as well as interfaces for creation of both simple and technology-enhanced items. They can also innovate with randomized variants of items, solution validation and customized feedback to students.

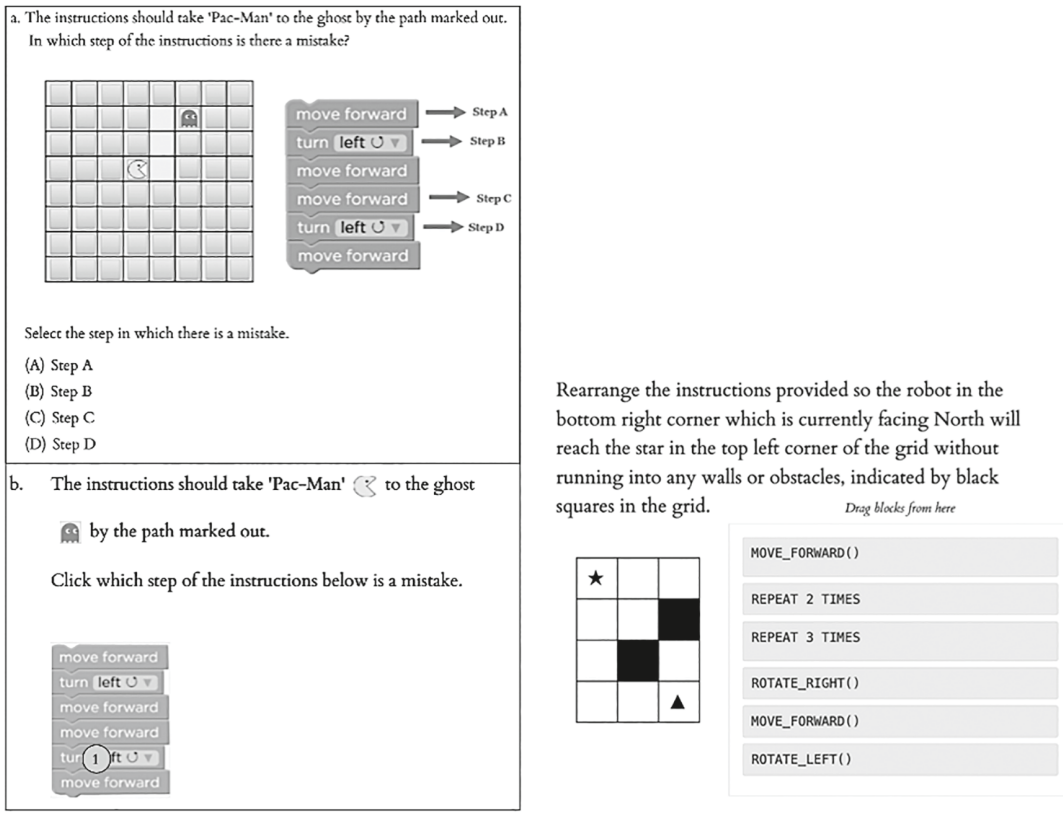

Figure 16.4 (a) An MCQ item from Computational Thinking Test (CTt) adapted into a point-and-click item on Edfinity.com and (b) a Parson's puzzle problem for AP CS Principles

16.5 Summary

The focus of this chapter has been primarily on formative assessment, or assessment for learning, which includes feedback, questioning, self-assessment opportunities, peer assessment and a range of other strategies in the context of computing. This is based on the innovative work of Black and Wiliam (1998).

The value of both peer and self-assessment, part of formative assessment, is supported in the research (Liu et al., 2001; Boud, Lawson and Thompson, 2013). Not only do learners receive valuable formative feedback, but it also gives them an opportunity to understand better the requirements or objectives of the task they are undertaking. The one concern to acknowledge is that the assessment by peers may not always be congruous with assessment by teachers. Peers may have a tendency to be generous with marks (Sajjadi et al., 2016).

In the computer science classroom, because much time is devoted to programming, it is tempting to assess the outcome of that task (Insa and Silva, 2015; Lobb and Harlow, 2016). However, assessing

the artefact produced by the task of programming has been shown to be less reliable than methods that assess the process of producing the artefact (Brennan and Resnick, 2012; Portelance and Bers, 2015).

An assessment opportunity is a task that may be set with the intention of using it for assessment. These may include a programming environment with the ability to generate feedback for the learner or the teacher. Parson's puzzles, a task where algorithm steps are ordered, can be accomplished in online environments (Ihantola and Karavirta, 2011) but can also be done with paper cut-outs. Rubrics are a fair and reliable (Barney et al., 2012) assessment tool, where learners can reflect on their own performance and judge how to reach a higher level (Mustapha et al., 2016). When learners produce concept maps, they reveal how they organize their knowledge and understanding. Teachers can interpret the maps to reveal students' misconceptions (Wei and Yue, 2016). MCQs can be used, as formative assessment tools, in several different ways in the classroom. Clicker technologies (Ribbens and National Science Teachers Association, 2007) and online systems bring a competitive environment to the activity of assessment while allowing the teacher to collect immediate feedback and respond immediately to misunderstandings and misconceptions (Kay and LeSage, 2009). However, to reap the most benefit from these systems, the students and teachers must use them as a tool, rather than just an engaging activity.

There is much research still to be conducted in the area of assessment in computer science education. Areas of particular interest include understanding how classroom dialogue can be leveraged to formatively assess computational thinking skills and understanding the extent to which peer and self-assessment in schools reflects the bias shown in higher education. Reproducing and testing these studies in a new context will lead to more effective teaching and learning.

Key points

- Many assessment methods can be used in conjunction to effectively formatively assess students in computing.
- Self- and peer assessment enable students to become familiar with the criteria used to assess them and thus become more aware of what they need to do to progress.
- Concept maps are useful to examine students' understanding of the relationship between key concepts and ideas.
- Automated tools for assessing programming are becoming more apparent with more new developments expected in the next few years.
- Multiple-choice questions and response systems using technology can give both teacher and student instant feedback about their progress and stimulate discussion about right and wrong answers to questions.

For further reflection

- For each of the assessment approaches listed in this chapter, make a list of example topics that would benefit from this type of assessment and devise some assessment opportunities.
- Focusing on a topic in computing that students might find difficult, reflect on the type of assessment that would enable a teacher to have a good understanding of what the student understood and any persistent misconceptions.

References

Alaoutinen, S., and Smolander, K. (2010), 'Student Self-Assessment in a Programming Course Using Bloom's Revised Taxonomy', in *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, 155–9, ACM.

Andrade, H. G. (2000), 'Using Rubrics to Promote Thinking and Learning', *Educational Leadership*, 57 (5): 13–19.

Barney, S., Khurum, M., Petersen, K., Unterkalmsteiner, M., and Jabangwe, R. (2012), 'Supporting Students Improve with Rubric-Based Self-Assessment and Oral Feedback', *IEEE Transactions on Education*, 55 (3): 319–25.

Barron, B., Martin, C., Roberts, E., Osipovich, A., and Ross, M. (2002), 'Assisting and Assessing the Development of Technological Fluencies: Insights from a Project-Based Approach to Teaching Computer Science', in *Proceedings of Computer Support for Collaborative Learning*.

Becker, K. (2003), 'Grading Programming Assignments Using Rubrics', *ACM SIGCSE Bulletin*, 35 (3): 253.

Black, P., and Wiliam, D. (1998), *Inside the Black Box: Raising Standards through Classroom Assessment*, London: GL Assessment.

Borda, E. J., Burgess, D. J., Plog, C. J., DeKalb, N. C., and Luce, M. M. (2009), 'Concept Maps as Tools for Assessing Students' Epistemologies of Science', *Electronic Journal of Science Education*, 13 (2): 160–85.

Boud, D., and Falchikov, N. (2007), *Rethinking Assessment in Higher Education*, London: Kogan Page.

Boud, D., Lawson, R., and Thompson, D. G. (2013), 'Does Student Engagement in Self-Assessment Calibrate Their Judgement Over Time?' *Assessment & Evaluation in Higher Education*, 38 (8), 941–56.

Brennan, K., and Resnick, M. (2012), 'New Frameworks for Studying and Assessing the Development of Computational Thinking', in *Paper presented at the Proceedings of the 2012 Annual Meeting of the American Educational Research Association*.

Clark, N. (2004), 'Peer Testing in Software Engineering Projects', in *Proceedings of the Sixth Australasian Conference on Computing Education*, 30, 41–8, Australian Computer Society, Inc.

Coca, D., and Slisko, J. (2013), 'Software Socrative and Smartphones as Tools for Implementation of Basic Processes of Active Physics Learning in Classroom: An Initial Feasibility Study with Prospective Teachers', *European Journal of Physics Education*, 4 (22): 8.

CSTA (n.d.), Standards. https://www.csteachers.org/page/standards.

Cutri, R., Marim, L. R., Cordeiro, J. R., Gil, H. A. C., and Guerald, C. C. T. (2016, 26 June), 'Kahoot, a New and Cheap Way to Get Classroom-Response Instead of Using Clickers', in *Paper presented at 2016 ASEE Annual Conference & Exposition*, New Orleans, LO. https://peer.asee.org/kahoot-a-new-and-cheap-way-to-get-classroom-response-instead-of-using-clickers.

Dell, K. A., and Wantuch, G. A. (2017), 'How-to-Guide for Writing Multiple Choice Questions for the Pharmacy Instructor', *Currents in Pharmacy Teaching and Learning*, 9 (1): 137–44. doi: http://dx.doi.org/10.1016/j.cptl.2016.08.036.

Denner, J., Werner, L., and Ortiz, E. (2012), 'Computer Games Created by Middle School Girls: Can They Be Used to Measure Understanding of Computer Science Concepts?' *Computers and Education*, 58 (1): 240–9. doi: 10.1016/j.compedu.2011.08.006.

Denny, P., Luxton-Reilly, A., and Simon, B. (2008, September), 'Evaluating a New Exam Question: Parsons Problems', in *Proceedings of the Fourth International Workshop on Computing Education Research*, 113–24.

Doppelt, Y. (2009), 'Assessing Creative Thinking in Design-Based Learning', *International Journal of Technology and Design Education*, 19 (1): 55–65. doi: 10.1007/s10798-006-9008-y.

Eugene, K., Stringfellow, C., and Halverson, R. (2016), 'The Usefulness of Rubrics in Computer Science', *Journal of Computing Sciences in Colleges*, 31 (4): 5–20.

Fotaris, P., Leinfellner, R., Mastoras, T., and Rosunally, Y. (2016), 'Climbing Up the Leaderboard: An Empirical Study of Applying Gamification Techniques to a Computer Programming Class', *Electronic Journal of e-Learning*, 14 (2): 94–110.

Freeman, L. A., and Urbaczewski, A. (2001), 'Using Concept Maps to Assess Students' Understanding of Information Systems', *Journal of Information Systems Education*, 12 (1): 3–9.

Giordano, D., Maiorana, F., Csizmadia, A. P., Marsden, S., Riedesel, C., Mishra, S., and Vinikienė, L. (2015, July), 'New Horizons in the Assessment of Computer Science at School and Beyond: Leveraging on the Viva Platform', in *Proceedings of the 2015 ITiCSE on Working Group Reports*, 117–47, ACM.

Grover, S. (2017), 'Assessing Algorithmic and Computational Thinking in K-12: Lessons from a Middle School Classroom', *Emerging Research, Practice, and Policy on Computational Thinking*, 269–88, Cham: Springer International.

Grover, S. (2021), 'A Framework for Formative Assessment and Feedback to Support Student Learning in CS Classrooms', in *Understanding Computing Education (vol. 1), Proceedings of the Raspberry Pi Foundation Research Seminar Series*. rpf.io/seminar-proceedings-2020.

Grover, S., and Twarek, B. (2021), 'Misconceptions as Learning Opportunities', *Workshop at the 2021 Computer Science Teachers' Association Conference*.

Grover, S., Pea, R., and Cooper, S. (2014, March), 'Promoting Active Learning & Leveraging Dashboards for Curriculum Assessment in an OpenEdX Introductory CS Course for Middle School', in *Proceedings of the First ACM Conference on Learning@ Scale Conference*, 205–6.

Grover, S., Pea, R., and Cooper, S. (2015), 'Designing for Deeper Learning in a Blended Computer Science Course for Middle School Students', *Computer Science Education*, 25 (2): 199–237.

Grover, S., Sedgwick, V., and Powers, K. (2020), 'Feedback through Formative Check-Ins', in S. Grover (ed.), Computer Science in K-12: An A to Z Handbook on Teaching Programming, Palo Alto, CA: Edfinity.

Grover, S., Carmichael, B., Venkatasubramaniam, S., and Twarek, B. (2022), 'Beyond MCQ: Designing Engaging, Autogradable Assessments for Supporting Teaching & Learning in K-12 Computer

Science', in *To be presented at the 53rd ACM Technical Symposium on Computing Science Education (SIGCSE'22),* ACM.

Hamm, R. W., Henderson, K. D., Repsher, M. L., and Timmer, K. L. (1983), 'A Tool for Program Grading: The Jacksonville University Scale', in *Proceedings of the 14th SIGCSE Technical Symposium on Computer Science Education*, 248–52, ACM.

Hofstein, A., and Lunetta, V. N. (2004), 'The Laboratory in Science Education: Foundations for the Twenty-First Century', *Science Education*, 88 (1): 28–54. doi: 10.1002/sce.10106.

Ihantola, P., and Karavirta, V. (2011). 'Two-Dimensional Parson's Puzzles: The Concept, Tools, and First Observations', *Journal of Information Technology Education. Innovations in Practice*, 10, 119–32.

Insa, D., and Silva, J. (2015), 'Semi-Automatic Assessment of Unrestrained Java Code: A Library, a DSL, and a Workbench to Assess Exams and Exercises', in *Paper presented at the Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, Vilnius, Lithuania.

Kay, R. H., and LeSage, A. (2009), 'Examining the Benefits and Challenges of Using Audience Response Systems: A Review of the Literature', *Computers & Education*, 53 (3): 819–27. http://dx.doi.org/10.1016/j.compedu.2009.05.001.

Liu, E. Z. F., Lin, S. S., Chiu, C. H., and Yuan, S. M. (2001), 'Web-Based Peer Review: The Learner as Both Adapter and Reviewer', *IEEE Transactions on Education*, 44 (3): 246–51.

Lobb, R., and Harlow, J. (2016), 'Coderunner: A Tool for Assessing Computer Programming Skills', *ACM Inroads*, 7 (1): 47–51. doi: 10.1145/2810041.

Luxton-Reilly, A. (2009), 'A Systematic Review of Tools That Support Peer Assessment', *Computer Science Education*, 19 (4); 209–32. https://doi.org/10.1080/08993400903384844.

Markham, K., and Mintzes, J. J. (1994), 'The Concept Map as a Research and Evaluation Tool: Further Evidence of Validity', *Journal of Research in Science Teaching*, 31 (1): 91–101.

Miller, N. E., and Peterson, C. G. (1980), 'A Method for Evaluating Student Written Computer Programs in an Undergraduate Computer Science Programming Language Course', *SIGCSE Bulletin*, 12 (4): 9–17.

Moen, P. (2009), 'Concept Maps as a Device for Learning Database Concepts', in *7th Workshop on Teaching, Learning and Assessment in Databases*, Birmingham, UK, HEA.

Morrison, B. B., Margulieux, L. E., and Guzdial, M. (2015, August), 'Subgoals, Context, and Worked Examples in Learning Computing Problem Solving', in *Proceedings of the Eleventh Annual International Conference on International Computing Education Research,* 21–9.

Mühling, A. (2016), 'Aggregating Concept Map Data to Investigate the Knowledge of Beginning CS Students', *Computer Science Education*, 26 (2–3): 176–91.

Murphy, L., and Tenenberg, J. (2005), 'Knowing What I Know: An Investigation of Undergraduate Knowledge and Self-Knowledge of Data Structures', *Computer Science Education*, 15 (4): 297–315.

Mustapha, A., Samsudin, N. A., Arbaiy, N., Mohammed, R., and Hamid, I. R. (2016), 'Generic Assessment Rubrics for Computer Programming Courses', *Turkish Online Journal of Educational Technology–TOJET*, 15 (1): 53–68.

Popham, W. J. (1997), 'What's Wrong—and What's Right—with Rubrics', *Educational Leadership*, 55: 72–5.

Portelance, D. J., and Bers, M. U. (2015), 'Code and Tell: Assessing Young Children's Learning of Computational Thinking Using Peer Video Interviews with ScratchJr', in *Paper presented at the Proceedings of the 14th International Conference on Interaction Design and Children*, Boston, MA.

Sajjadi, M. S., Alamgir, M., and von Luxburg, U. (2016), 'Peer Grading in a Course on Algorithms and Data Structures: Machine Learning Algorithms Do not Improve over Simple Baselines', in *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*, 369–78, ACM.

Sanders, K., Ahmadzadeh, M., Clear, T., Edwards, S. H., Goldweber, M., Johnson, C., Lister, R., McCartney, R., Patitsas, E., and Spacco, J. (2013, June), 'The Canterbury QuestionBank: Building a Repository of Multiple-Choice CS1 and CS2 Questions', in *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education-Working Group Reports*, 33–52.

Seiter, L. (2015), 'Using SOLO to Classify the Programming Responses of Primary Grade Students', in *Paper presented at the Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, Kansas City, MI.

Sitthiworachart, J., and Joy, M. (2003, July), 'Web-Based Peer Assessment in Learning Computer Programming', in 3rd IEEE International Conference on Advanced Learning Technologies, Proceedings, 180–4, Athens: IEEE.

Topping, K. J (2009), 'Peer Assessment', *Theory into Practice*, 48 (1): 20–7.

Tseng, S. C., and Tsai, C. C. (2007), 'On-line Peer Assessment and the Role of the Peer Feedback: A Study of High School Computer Course', *Computer & Education*, 49: 1161–74.

Wang, A. I., Zhu, M., and Sætre, R. (2016, January), 'The Effect of Digitizing and Gamifying Quizzing in Classrooms', in *Paper presented at the 10th European Conference on Games Based Learning*.

Wei, W., and Yue, K. B. (2016), 'Using Concept Maps to Assess Students' Meaningful Learning in IS Curriculum', in *Conference on Information Systems and Computing Education EDSIG*, Las Vegas.

Werner, L., Campe, S., and Denner, J. (2012), 'Children Learning Computer Science Concepts via Alice Game-Programming', in *Paper presented at the Proceedings of the 43rd ACM technical symposium on Computer Science Education*, Raleigh, NC.

Part 4

A Focus on Programming

17

Introduction to Part 4

Sue Sentance

With the introduction of computer science in schools, we have probably all seen the media and others become rather fixated with 'coding', as if computer science were only about coding and nothing else. There are two problems with this common overgeneralization. The first is the omission of all the other aspects of computer science that are taught in school, including networking, computer architecture, human–computer interaction, data and information, and the impact of technology on society. Second is the assumption that coding is the same as programming, which it is not. Where coding assumes that an algorithm is to be translated into program code and then executed, programming includes the whole process of designing a solution, creating an algorithm and then coding.

We can see that programming is a substantial topic and has frequently been reported as difficult to teach and learn; in addition, there has been a lot of research into how we learn to program over several decades. In this second edition, we decided to increase the focus on programming to capture the full breadth of the subject, including oft-neglected design (Chapter 19) as well as the new and exciting topic of epistemic programming (Chapter 22). It's exciting to be able to delve a little deeper, and hopefully, readers will find this useful.

In Chapter 18, Michael E. Caspersen starts us thinking about the teaching of programming. There is a long history in computing education research around the teaching of programming, from the 1970s or even earlier. In this chapter we learn about a number of key principles for teaching programming, covering a number of strategies including the value of worked examples and stepwise improvement as useful approaches for developing a secure understanding of programming concepts and skills. Reflecting on programming at this level may give you some surprising insights into your understanding of programming: demonstrating that subject and pedagogical knowledge is complex and cannot be expressed as recipes.

In Chapter 19, Jane Waite draws on her research with primary teachers to consider the potential uses of design, the difficulties of design and what underlying concepts might help teachers to support the teaching and learning of design at the primary level. It's an important aspect of programming and often neglected

In Chapter 20, Juha Sorva gives an overview on typical misconceptions in the computing classroom and gives some advice for misconception-sensitive teaching. Students do not enter the classroom as blank slates, ready to be filled with concepts from computer science – instead they are on a learning journey and may hold alternate conceptions on the way: it's useful for teachers to be able to uncover them and discuss concepts with learners.

Chapter 21 moves us on to specific practice in the school programming classroom. In this chapter, written by Jane Waite and me, we look at a number of specific programming strategies for teaching, the different contexts for learning programming and how we can support learners.

Finally, in Chapter 22, the authors switch their focus to a brand new perspective on programming. In this chapter, Sven Hüsing, Carsten Schulte and Felix Winkelnkemper define epistemic programming as 'the acquisition of new insights and the expression and representation of ideas'. This view of programming offers us a way of developing and presenting new insights and supports the creation of meaning. Analysing epistemic programming via people, project, process and product, the authors highlight how the students will be able to use programming to better understand the world and as a means for self-expression. In this sense, the analysis of large data sets is a typical application of epistemic programming.

18

Principles of Programming Education

Michael E. Caspersen

Chapter outline

Chapter synopsis

The defining characteristics of the computer is its programmability, and programming is the essence of computing/informatics. Indeed, computing is much more than programming, but programming – the process of expressing one's ideas and understanding of the concepts and processes of a domain in a form that allows for execution on a computing device without human interpretation – is essential to computing.

Teaching and learning programming is not easy; in fact, it is considered one of the grand challenges of computing education. In this chapter, we describe the nature of the challenge, and we provide a dozen teaching principles to help overcome the challenge.

18.1 Introduction

Writing a chapter about the principles of teaching of programming is an intriguing task but for many reasons also challenging – an entire book could be written on the subject.

This decision has advantages and disadvantages. An advantage is that the chapter is applicable regardless of which language technology you as a teacher intend to use. A disadvantage is the lack of concrete examples, expressed in a specific programming technology that you can apply directly in your teaching.

Section 18.2 describes the challenge of teaching programming. In section 18.3 – the heart of the chapter – we present a dozen principles that can help overcome some of the challenges of teaching and learning programming.

18.2 Teaching and learning programming is a grand challenge

In some ways, programming education has changed dramatically over the past more than fifty years. We have experienced a rich and successful development in programming language technologies and an accompanying development of teaching practices.

However, in other ways, things have not changed that much, and it is still the case that typical introductory programming textbooks devote most of their content to presenting knowledge about a particular language (Robins, Rountree and Rountree, 2003).

Exposing students to the process of programming is merely implied but not explicitly addressed in texts on programming, which appear to deal with 'program' as a noun rather than as a verb.

But teaching programming is much more than teaching a programming language. Knowledge about a programming language is a necessary but far-from-sufficient condition for learning the practice of programming. Students also need knowledge about *the programming process*, that is, how to *develop* programs, and they need to extend that knowledge into *programming skills*.

David Gries (1974: 82) pointed this out already in 1974, when he wrote the following:

> Let me make an analogy to make my point clear. Suppose you attend a course in cabinet making. The instructor briefly shows you a saw, a plane, a hammer, and a few other tools, letting you use each one for a few minutes. He next shows you a beautifully-finished cabinet. Finally, he tells you to design and build your own cabinet and bring him the finished product in a few weeks. You would think he was crazy!

Clearly, cabinet making cannot be taught simply by teaching the tools of the trade and demonstrating finished products, but neither can programming.

Nevertheless, judging by the majority of past as well as contemporary textbooks, this is what is being attempted. In Kölling (2003), a survey of thirty-nine major selling textbooks on introductory programming was presented. The overall conclusion of the survey was that all books are structured according to the language constructs of the programming language; the process of program development is often merely implied rather than explicitly addressed.

A typical structure of a section on a specific language construct (e.g. the while loop) is the presentation of a problem followed by a presentation of a program to solve that problem and a discussion of the program's elements. From the viewpoint of a student, the program was developed in a single step, starting from a problem specification and resulting in a working solution.

This pattern of introducing material creates – unintentionally – the illusion that programming is trivial and straightforward. The fact that we all, when we start addressing a problem, start with incomplete and incorrect programs, which we then gradually modify by extending, refining and restructuring our implementation until we arrive at an acceptable solution, seems to be swept under the carpet as if it was an embarrassing secret that must not be mentioned. While the ultimate solution to the problem is explained in detail, the process – how we go about developing the solution – is almost entirely neglected in textbooks and beginners' courses (Caspersen and Kölling, 2009). Essentially, programming is one of the best-kept secrets of programming education!

In a time where computing/informatics education is becoming general education for all and students don't choose to learn programming out of personal interest, the challenge not only persists but also is reinforced.

According to du Boulay (1989) the difficulties of novices learning programming can be separated into five partially overlapping areas:

- Orientation: finding out what programming is for
- The notional machine: understanding the general properties of the machine that one is learning to control
- Notation: problems associated with the various formal languages that have to be learned, mastering both syntax and semantics
- Structure: the difficulties of acquiring standard patterns or schemas that can be used to achieve small-scale goals such as computing the sum using a loop or implementing a 0..* association between two classes
- Pragmatics: the skill of how to specify, develop, test and debug programs using whatever tools are available

The good news is that there are some relatively simple and effective didactical principles that help alleviate the challenge; we organize the principles in four categories:

1. Progression
   - **1.1.** Be application oriented
   - **1.2.** Let students progress from consumer to producer (use–modify–create)
   - **1.3.** Organize the progression in terms of complexity of tasks, not complexity of language constructs
2. Examples
   - **2.1.** Provide exemplary examples
   - **2.2.** Provide worked examples
   - **2.3.** Establish motivation through passion, play, peers and meaningful projects
3. Process
   - **3.1.** Reveal process and pragmatics
   - **3.2.** Provide scaffolding through stepwise self-explanations
   - **3.3.** Apply and teach incremental development through stepwise improvement (i.e. extend, refine, restructure)

4. Abstraction
   - **4.1.** Reinforce specifications
   - **4.2.** Reinforce patterns
   - **4.3.** Reinforce models and conceptual frameworks (program *into* a language)

These teaching principles are described in detail in the next section. However, while the principles help overcome many aspects of the challenge of teaching and learning programming, it still persists (Gries, 2002: 5):

> Programming is a skill, and teaching such a skill is much harder than teaching physics, calculus or chemistry. People expect a student coming out of a programming course to be able to program any problem. No such expectations exist for a calculus or chemistry student. Perhaps our expectations are too high.
>
> Compare programming to writing. In high school, one learns about writing in several courses. In addition, every college freshman takes a writing course. Yet, after all these courses, faculty members still complain that students cannot organise their thoughts and write well! In many ways, programming is harder than writing, so why should a single programming course produce students who can organise their programming thoughts and program well.

Is writing hard? Is teaching writing hard? Well, clearly it depends a great deal upon what you are trying to (teach your students to) write. Writing a novel, a textbook or a dissertation is very hard, and it is hard to teach how to do so. Writing an article, a feature or a report is also fairly hard and hard to teach. Writing a birthday greeting, a to-do list or a text message is much easier and requires very little instruction apart, of course, from learning the basics of (reading and) spelling.

Learning programming is not very different from learning writing. Most of the time, programming is creative and fun. However, like writing, programming is not trivial, and we teachers must embrace the challenge with enthusiasm as well as with a humble attitude and reasonable expectations. The most important aspect of learning programming as well as writing is to program/write things that matter to us.

18.3 Principles for teaching programming

In this section, we describe twelve teaching principles, or didactical principles, organized in four categories. The principles, which are all backed by research and experience, can help overcome some of the challenges of teaching and learning programming.

Progression

Principle 1.1: Be application oriented

Traditionally, introductory programming courses apply a bottom-up approach, in the sense that students are introduced to basic and foundational concepts and expected to master these before more advanced concepts and principles are introduced. Hence, in a traditional programming

course, students are often trained in constructing a trivial program as the very first activity, and then later on they are trained in adding more layers of complexity to a system in terms of user interfaces, databases, and so on. For the technically inclined students, this may be a feasible approach, but for a more general audience, this could pose severe motivational problems, as we are dealing with a wider range of students with much more diverse interests and backgrounds.

There is an even more important reason why a traditional bottom-up approach is fallible. For a general audience, we are not aiming at developing detailed and concrete competences in a specific programming technology; instead, we are aiming at developing interest, critical thinking, creativity and broader skills in programming and computational thinking and practice. Therefore, we recommend an application-oriented top-down approach. This implies to start various teaching activities by introducing well-known or familiar applications, which is then split apart for conceptual and/or technical examination, evaluation and modification.

For motivational reasons, we recommend applications based on the criteria that they must matter to students in the relevant age range, applications which they find interesting to use and hopefully to examine and improve. Examples could include pedagogical lightweight versions of Facebook, iTunes/Spotify, YouTube, Twitter, Blogs, Photoshop, Instagram and similar applications. Or it could be something embedded in a physical context based on Internet of Things, for example wearables and smart clothes. But these are just examples; in general, the choice of application types depends on the specific context and target group (Caspersen and Nowack, 2013).

For a specific technology like Scratch, there are a great number of approaches and domains that have inspired educators and researchers to develop teaching materials, for example a data-driven approach (Dasgupta and Resnick, 2014), a creativity- and maker-driven approach (Brennan, Balch and Chung, 2014) and a computing concept-driven approach (Armoni and Ben-Ari, 2010; Meerbaum-Salant, Armoni and Ben-Ari, 2013).

Principle 1.2: Let students progress from consumer to producer (use–modify–create)

Pattis (1990) introduces the *call before write* approach to teaching introductory programming, arguing that it 'allows students to write more interesting programs early in the course and it familiarizes them with the process of writing programs that call subprograms; so it is more natural for them to continue writing well-structured programs after they learn how to write their own subprograms'.

Meyer (1993) introduces the notion of the *inverted curriculum* as follows: 'This proposal suggests a redesign of the teaching of programming and other software topics in universities on the basis of object-oriented principles. It argues that the new "inverted curriculum" should give a central place to libraries, and take students from the reuse consumer's role to the role of producer through a process of "progressive opening of black boxes." '

) briefly mentions the notion of *consuming before producing* by providing three specific examples. One example is as follows: 'BlueJ allows beginning with an object "system" with just one class where students just interactively use instances of this class (they *consume* the notion of interacting with an object via its interface). *Producing* the possibility of interacting with an object, on the other

hand, requires more knowledge about class internals and should thus be done after the principle of interaction with objects is well understood.'

The *consume before produce* principle is applicable to a wide number of topics, for example code, specifications, class libraries and frameworks/event-driven programming.

Code: The principle may be applied with respect to the way students write code at three levels of abstractions: method level, class (or module) level and modelling level as follows: (1) *Use methods* (as indicated above, BlueJ allows interactive method invocation on objects without writing any code). At this early stage, students can perform experiments with objects in order to investigate the behaviour and determine the actual specification of a method. (2) *Modify methods* by altering statements or expressions in existing methods. (3) *Extend methods* by writing additional code in existing methods. (4) *Create methods* by adding new methods to an existing class. This may also be characterized as *extend class*. (5) *Create class/module* by adding new classes/modules to an existing model. This may also be characteriszd as *extend model*. (6) *Create model* by building a new model for a system to be implemented.

Specifications: Specifications and assertions can be expressed in many ways, for example as Javadoc, test cases, general assertions in code, loop invariants, class invariants and system invariants (constraints in the class model, for instance, a specific multiplicity on a relation between two classes). In all cases, students are gradually exposed to reading and comprehending specifications prior to producing specifications themselves.

Class libraries: Not many years ago, the standard syllabus for introductory programming courses encompassed implementation of standard algorithms for searching and sorting as well as implementation of standard data structures such as stacks, queues, linked lists, trees and binary search trees.

These days, standard algorithms and data structures are provided in class libraries, ready to be used by programmers. By using class libraries that provide advanced functionality, students can do much more interesting things more quickly. Also, experience as consumer presumably motivates learning more about the principles and theory behind advanced data structures and packages for distributed programming, and so on.

Consequently, algorithms and data structures are one of the areas where we can sacrifice material in order to find room for all the new things that make up a modern introductory programming course.

Frameworks/event-driven programming: Sometimes even using a piece of software can be a daunting task. Frameworks are examples of such complex pieces of software.

Frameworks may constitute a part of an introductory programming course, but in order to ease comprehension of a complex frameworks with call-back methods (inversion of control), it helps to provide a stepping stone in the form of a small and simple framework, which students consume by making a few simple instantiations. After the road has been paved, you may provide a more general taxonomy for frameworks/event-based programming, which is now more easily understood and grasped in the context of the simple toy framework. With the concrete experiences and the taxonomy in the bag, students are prepared to embark on using more complex frameworks.

Christensen and Caspersen (2002) provide a more thorough discussion of an approach to teaching frameworks and event-based programming in introductory programming courses.

Use–Modify–Create: Lee et al. (2011) describe how computational thinking takes shape for middle and high school youth in the United States. They propose using a three-stage progression for

engaging youth in computational thinking. This progression, called use–modify–create, describes a pattern of engagement that was seen to support and deepen youths' acquisition of computational thinking (see also Chapter 3, 'Perspectives on Computing Curricula').

Caspersen and Bennedsen (2007) describe a similar three-stage progression for working with programs, called use–extend–create. Caspersen and Nowack (2013) also describe use–modify–create as a specialization of consumer-to-producer. A more elaborate or fine-grained version is use–alter–test–modify–assess–refine–evaluate–create.

conclude that the use–modify–create progression is a useful framework for educators and researchers looking at how computational thinking develops and how that development can be supported.

Principle 1.3: Organize the progression in terms of complexity of tasks, not complexity of language constructs

Typically, progression in introductory programming courses is dictated by a bottom-up treatment of the language constructs of the programming language being used, and this is the way most textbooks are structured. We hold as a general principle that the progression in the course is defined in terms of the complexity of the worked examples presented to the students and the corresponding exercises and assignments.

Example: Progression

A concrete example of progression in terms of complexity of task is provided by the classical Turtle Graphics, developed by Seymour Papert. Students can start by making utterly simple programs/drawings and then gradually, based on the student's ambitions and skills, progress to make more advanced and ultimately quite complicated programs/drawings. Thus, Turtle Graphics exhibit what Papert called 'low floor, high ceiling'; that is, it allows for easy entrance without restricting the power of expression.

Another example is to let the progression be defined in terms of complexity of program structure or architecture (e.g. classes and their relationship) by starting with very simple programs with simple functionality and only few components with very simple relationships and then progress to more complex programs with increasingly complex structure and richer functionality. This could be starting with a program with just one component (representing, say, a die, a date, a person, an image, a song, etc.); then working with programs with two components (representing, say, a die and a die cup, a date and a clock, a person and a party, a song and a playlist, etc.) and then go on to working with programs with many components and more complex structures (e.g. a game, music player, an image processing app, etc.). See Bennedsen and Caspersen (2004b) for further details.

It is of course possible to turn things around and start out using a more complex app, then zoom in and work on a smaller part, perhaps just one component, while modifying this.

Examples

Principle 2.1: Provide exemplary examples

Examples are important teaching tools. Research in cognitive science confirms that 'examples appear to play a central role in the early phases of cognitive skill acquisition' (VanLehn, 1996). An alternation of worked examples and problems increases the learning outcome compared with just solving more problems (Sweller and Cooper, 1985; Trafton and Reiser, 1993).

Students generalize from examples and use them as templates for their own work. Examples must therefore be easy to generalize and consistent with current learning goals.

By perpetually exposing students to 'exemplary' examples, desirable properties are reinforced many times. Students will eventually recognize patterns of 'good' design and gain experience in telling desirable from undesirable properties.

With carefully developed examples, teachers can minimize the risk of misinterpretations and erroneous conclusions, which otherwise can lead to misconceptions. Once established, misconceptions can be difficult to resolve and hinder students in their further learning (Clancy, 2004; Ragonis and Ben-Ari, 2005). See also Chapter 20, in which Juha Sorva details many different misconceptions.

A good example must be understandable by students. Without 'understanding', knowledge retrieval works only on an example's surface properties, instead of on its underlying structural and conceptual properties (Trafton and Reiser, 1993; VanLehn, 1996).

A good example must furthermore effectively communicate the concept(s) to be taught. There should be no doubt about what exactly is exemplified. The structural form of information affects the form of the knowledge encoded in human memory. Conceptual knowledge is improved by best examples and by expository examples, where the best example represents an average, central or prototypical form of a concept. To minimize cognitive load, an example should furthermore exemplify only one new concept (or very few) at a time.

The two example properties, (1) understandable by students and (2) effectively communicating the concept(s) to be taught, might seem obvious. However, the recurring discussions about the harmfulness or not of certain common examples show that there is quite some disagreement in the teaching community about the meaning of these properties. For further details, including many more references, see Börstler, Caspersen and Nordström (2007, 2016) and Börstler, Nordström and Paterson (2011).

Principle 2.2: Provide worked examples

Studies of students in a variety of instructional situations have shown that students prefer learning from examples rather than learning from other forms of instruction. Students learn more from studying examples than from solving the same problems themselves.

A worked example (WE), consisting of (1) a problem statement and (2) a procedure for solving the problem, is an instructional device that provides a problem a solution for a learner to study (Atkinson et al., 2000; Chi et al., 1989; LeFevre and Dixon, 1986). WEs are meant to illustrate how similar problems might be solved, and WEs are effective instructional tools in many domains, including programming. WEs combined with faded guidance are particularly effective (Caspersen and Bennedsen, 2007).

Atkinson et al. (2000) emphasize three major categories that influence learning from worked examples; Caspersen and Bennedsen (2007) present the categories as how-to principles of constructing and applying examples in education: (1) How to construct examples, (2) how to design lessons that include examples and (3) how to foster students' thinking process when studying examples. We return to the latter when we discuss Principle 3.2: Provide scaffolding through stepwise self-explanations.

Bennedsen and Caspersen (2004a) illustrate implicitly how WEs can be used to teach programming using a systematic, model-based programming process. Caspersen and Bennedsen (2007) present an instructional design for an introductory programming course based on a thorough use of WE, and Caspersen (2007) provides an overview of WE literature related to programming education as well as a survey of the related cognitive load theory (CLT).

Principle 2.3: Establish motivation through passion, play, peers and meaningful projects

Mitch Resnick and his research group at the Lifelong Kindergarten at the MIT Media Lab have been developing new technologies, activities and strategies to engage young people in creative learning experiences (Resnick, 2014). Their approach is based on four core elements, sometimes called the Four P's of Creative Learning:

- *Projects*. People learn best when they are actively working on meaningful projects – generating new ideas, designing prototypes, refining iteratively.
- *Peers*. Learning flourishes as a social activity, with people sharing ideas, collaborating on projects and building on one another's work.
- *Passion*. When people work on projects they care about, they work longer and harder, persist in the face of challenges and learn more in the process.
- *Play*. Learning involves playful experimentation – trying new things, tinkering with materials, testing boundaries, taking risks, iterating again and again.

In this example, WE and faded guidance is applied in five stages as follows:

1. In a video, present development of a player with two components, Playlist and Track. (A complete WE)
2. In a lecture, present (a partial) development of a similar example, that is, with the same structure but different cover stories. This could be, say, a simple banking system with components Account and Transaction. (A partial WE)
3. In a lab session, students use, modify and extend both examples.
4. In an exercise, students extend the player by adding an Image component that allows several images to be displayed when a Track is played.
5. In an assignment, students implement a similar system (again, similar structure, but different cover story), say, a notebook app with Notes, Keywords, Contacts, and so on.

Process

As mentioned in the first section, the process of programming is one of the best-kept secrets of programming education.

Thus, students are left on their own to find their process of programming. Instead of leaving the students on their own, we as educators must help students to develop a systematic process to learning programming; and we must provide guided tours to proper program development.

This section deals with the process from three perspectives: how to reveal the programming process, how to facilitate process-based self-explanations and how to conceptualize the programming process as consisting of three independent (but of course related) activities: extension, refinement and restructuring – stepwise improvement.

Principle 3.1: Reveal process and pragmatics

Revealing the programming process to beginner students is important, but traditional static teaching materials such as textbooks, lecture notes, blackboards, slide presentations, and so on are insufficient for that purpose. They are useful for the presentation of a product (e.g. a finished program), but not for the presentation of the dynamic process used to create that product.

In addition, the use of traditional materials has another drawback. Typically, they are used for the presentation of an ideal solution that is the result of a non-linear development process. Like others (Soloway, 1986; Spohrer and Soloway, 1986), we consider this to be problematic, because it will inevitably leave the students with the false impression that there is a linear and direct 'royal road' from problem to solution.

This is very far from the truth, but the problem for novices is that when they see their teacher present clean and simple solutions, they think they themselves should be able to develop solutions in a similar way. When they realize they cannot do so, they blame themselves and feel incompetent. Consequently, they will lose self-confidence and, in the worst case, their motivation for learning to program.

Therefore, we must also teach about the programming process. This can include the task of using tools and techniques to develop the solution in a systematic, incremental and typically non-linear way. An important part of this is to expound and to demonstrate that

- many small steps are better than a few large ones
- the result of every little step should be tested
- prior decisions may need to be undone and code refactored
- making errors is common also for experienced programmers
- compiler errors can be misleading/erroneous
- online documentation for class libraries provides valuable information and
- there is a systematic, however non-linear, way of developing a solution for the problem at hand

We cannot rely on the students to learn all of this by themselves, but by using an apprenticeship approach, we can show them how to do it. WEs are highly suitable for this purpose (see principle 2.2), and they can be effectively communicated via videos. For many more details on this, see Bennedsen and Caspersen (2005).

Principle 3.2: Provide scaffolding through stepwise self-explanations

Self-explanations provide guidance regarding the way that students can study and understand instructional material. Clark, Nguyen and Sweller (2005) define SE as 'a mental dialog that learners have when studying a worked example that helps them understand the example and build a schema from it'. According to Chiu and Chi (2014), the activity of self-explaining promotes learning through the elaboration of information being studied, associating this new information with learners' prior knowledge, making inferences and connecting two or more pieces of the given information.

The benefits of self-explanations were first shown by Chi et al. (1989). They found that good students' explanations provided justifications for steps in the examples and related those steps to the concepts presented in the instructional material. Those students also monitored their understanding while studying the examples.

More information on self-explanation and stepwise self-explanation (a specialization of self-explanation related to stepwise improvement) can be found in Caspersen (2007) and Aureliano, Tedesco and Caspersen (2016).

Principle 3.3: Apply and teach incremental development through stepwise improvement (i.e. extend, refine, restructure)

In traditional stepwise refinement (Dijkstra, 1969; Wirth, 1971; Back, 1978, 1988; Morgan, 1990), programming is regarded as the one-dimensional activity of refining abstract programs (i.e. programs containing non-executable specifications) to concrete programs (i.e. executable code) through a series of behaviour-preserving program transformations. The fundamental assumption of traditional stepwise refinement is that the complete specification, or the requirements, is known and addressed from the outset. Typically, stepwise refinement is described as a strict top-down process of programming.

Programming by stepwise improvement (Caspersen, 2007), on the other hand, is characterized as an explorative activity of discovery and invention that takes place in the three-dimensional space of *extension*, *refinement* and *restructuring*. Extension is the activity of extending the specification to cover more (use) cases; refinement is the activity of refining abstract code to executable code to meet the current specification, and restructure is the activity of improving non-functional aspects of a solution without altering its observable behaviour, such as design improvements through refactoring, efficiency optimizations or portability improvements.

A very simple example of stepwise improvement is the development of an app that can show a date and advance to the next/previous date by pushing dedicated buttons in the user interface. Such an app can be developed according to stepwise improvement in the following extension steps:

1. Construct (part of) the user interface, no functionality
2. Make the app work except for the last day of a month (assume thirty days in every month and ignore leap years)
3. Make the app work for a variable number of days per month
4. Make the app work for leap years (except centuries)

5. Make the app work for centuries (except four-centuries)
6. Make the app work for four-centuries

By breaking the problem down like this and developing the program in a number of increments/iterations where the specification is gradually extended, the programming task becomes much more manageable. Also, it allows for a success/celebration every time a new version is finished. There are many pedagogical advantages of organizing students' programming process according to stepwise improvement.

From the stepwise improvement framework, Caspersen and Kölling have designed a novice's process of (object-oriented) programming called STREAM (Caspersen and Kölling, 2009).

Abstraction

Principle 4.1: Reinforce specifications

In programming, it is essential, at many levels of abstraction, to be able to distinguish and separate *what* a (part of a) program does from *how* it does it. A description of what a program part does is called a specification; the implementation, that is, the actual code of the program part, is the ultimate description of how it does it. Typically, one specification has many implementations; that is, there may be many concrete ways to obtain a desired outcome (to meet a specification).

A specification may be expressed as a name of a function, as a comment in natural text or in a more formal way. The concrete syntactic expression of a specification is not essential; it is the notion of specification itself, and the ability to separate specification from implementation, that is essential. As Pattis (1990) points out: 'the linguistic ability to cleanly separate a subprogram's specification from its implementation' is required in order to practice the 'call before write' approach.

We therefore hold as principle that the notion of specification is treated as a first-class citizen in introductory programming courses. See Caspersen (2007) for more on this.

Principle 4.2: Reinforce patterns

A pattern captures and describes (the essence of) a recurring structure or process in a given domain. In music, there are patterns of chords that, with minor or major variations of the melody, are used again and again (search the web for 'three chords' or 'four chords' and see for yourself). This is also the case in programming, where programming patterns are used again and again to obtain variations of essentially the same structure or process.

The fundamental motivation for a pattern-based approach to teaching programming is that patterns capture chunks of programming knowledge. According to cognitive science and educational psychology, explicit teaching of patterns reinforces schema acquisition as long as the total cognitive load is 'controlled'.

We reinforce patterns at different levels of abstraction including elementary patterns, algorithm patterns and design patterns, but equally importantly, we provide a conceptual framework for object orientation that qualifies modelling and programming and increases transfer. Furthermore, we

stress coding patterns for standard relations between classes (Knudsen and Madsen, 1988; Madsen et al., 1993; East et al., 1996; Muller, 2005; Caspersen, 2007; Caspersen and Bennedsen, 2007).

Principle 4.3: Reinforce models and conceptual frameworks (program into a language)

The so-called Sapir-Whorf hypothesis from linguistics states that *language defines the boundaries of thought*. Programming languages are artificial and simple languages, and if a programmer's thoughts are nurtured only via the constructs of a specific programming language, these thoughts will be severely constrained and have very limiting boundaries.

All programming languages have limitations, and we overcome these limitations by thinking and designing in terms of richer and more appropriate concepts and structures, which we simulate in the technology at hand.

In the early days of assembly programming, programmers used jump and compare instructions to *simulate* selection and iteration. This was also the case for the earliest high-level languages with go-to statements as the only control structure. Then, control structures for selection and iteration were developed.

Similarly, in the early days of programming, there was no programming language support for arrays (lists) and records (tuples). Consequently, these had to be simulated through careful and minute programming activities.

As long as there was no support for subprograms but the notion was conceived, programmers had to simulate call and return (and, in the general case, maintain not one but a stack of return addresses).

When there was no support for classes, but the notion of abstract data type was conceived, programmers had to simulate this, again through structured and minute programming activities.

The historic development of programming languages can be viewed as a constant interplay between programming language constructs and architectural abstractions. Limitations in programming languages generate new concepts, architectural abstractions, which can be simulated in existing programming languages. Gradually, these architectural abstractions find their way into mainstream languages but only to generate more advanced needs and foster new architectural abstractions.

In object-oriented programming, UML and similar modelling languages provide a richer conceptual framework than most object-oriented programming languages support. For example, the notion of association (a special relation between program components) is not directly supported but has to be simulated through structured and minute programming activities. In principle, this is exactly the same situation as simulation of a while loop with go-to statements.

When teaching programming, it is important to provide a conceptual framework, which is richer than the concrete programming language being used. Programming should not be done *in* a language, but *into* a language.

Conceptual modelling. As a general foundation for informatics/computing, we recommend a general introduction to conceptual modelling. Unfortunately, there are no introductory texts on the subject, but Kristensen and Østerbye (1994: 83–6) provide a nice discussion on the subject;

the authors present the topic in the context of object-oriented programming languages, but it has much wider applications and implications.

The exposition in Kristensen and Østerby, which originates from Madsen, Møller-Perdersen and Nygaard (1993), presents a model of abstraction consisting of three abstraction processes (and their inverses): classification (exemplification), aggregation (decomposition) and generalization (specialization). The model may not be complete, but it has shown its applicability in many cases, including data modelling and object-oriented modelling, and has the potential to become a much more general framework for informatics and computational thinking. The current focus on computational thinking emphasizes abstraction and decomposition as major aspects (); however, the above-mentioned framework for conceptual modelling provides a richer and more general approach.

Sowa (1984) provides a broader approach to the topic but with a different perspective.

There are many learning-theoretic arguments for adopting a conceptual framework approach, for example a model-driven approach, to programming. We provide two (for more details, see Caspersen, 2007):

1. Because of their generic nature, the abstract models directly support schema creation and transfer:

> Well-designed learning environments for novices provide *metacognitive managerial guidance* to focus the students' attention and *schema substitutes* by optimizing the limited capacity of working memory in ways that free working memory for learning. Good instruction will segment and sequence the content in ways that reduce the amount of new information novices must process at one time and, as much as possible, reinforce domain patterns to support schema acquisition and improve learning.

2. Variation of form (e.g. cover story) can help novices realize that there is a many-to-one relationship between form and problem type: when students see a variety of cover stories used for identical or similar structures (of class models), they are more likely to notice that surface features are insufficient to distinguish among problem types and that problem categorization according to structural similarities (patterns) is imperative to enable reuse of solution schemas (Quilici and Mayer, 1996).

Models provide an excellent overview and generic approach to introductory programming. If pedagogical development tools more completely supported integration of code and UML-like models, we conjecture that the effect would be even better.

Example: Elementary patterns

Elementary patterns, that is, patterns for elementary program structure, exist in a number of variations, for example roles of variables (Sajaniemi, 2008), selection patterns (Bergin, 1999) and loop patterns (Astrachan and Wallingford, 1998).

Roles of variables represent programming knowledge that can be explicitly taught to students and which are easy to adopt in teaching. A number of roles for variables was identified and described by Sajaniemi (2008), for example fixed value, organizer, stepper, most-recent holder, one-way flag, most–wanted holder and so on.

Bergin (1999) divides selection patterns in three categories: basic selection patterns, strategy patterns and auxiliary patterns. Examples of basic selection patterns are Whether or Not and Alternative Action, examples of strategy patterns are Short Case First and Default Case First and examples of auxiliary patterns are Positive Condition and Function for Positive Condition.

Astrachan and Wallingford (1998) present a number of loop patterns related to sweeping over a (linear) collection of data. Examples of elementary loop patterns are Linear Search, Guarded Linear Search, Process All Items and Loop and a Half.

Key points

- Teaching and learning programming is a grand challenge
- In a time where computing/informatics education is becoming general education for all and students don't choose to learn programming out of personal interest, the challenge not only persists but also is reinforced
- Exposing students to the process of programming is merely implied but not explicitly addressed in texts on programming, which appear to deal with 'program' as a noun rather than as a verb
- Some relatively simple and effective teaching principles can help alleviate and overcome the challenge:
  - Progression: Be application-oriented; let students progress from consumer to producer (use-modify-create); organize progression in terms of complexity of task, not complexity of language constructs
  - Examples: Provide exemplary examples; provide worked examples; establish motivation through passion, play, peers and meaningful projects
  - Process: Reveal process and pragmatics; provide scaffolding through stepwise self-explanations; apply and teach incremental development through stepwise improvement (i.e. extend, refine, restructure).
  - Abstraction: Reinforce specifications; reinforce patterns; reinforce models and conceptual frameworks (program *into* a language)

For further reflection

- The use of examples is important in helping students to improve their programming skills. Consider how worked examples could be used in your classroom to support a particular programming topic.
- Considering the principles presented in this chapter, how do you think these could be incorporated at different points with young, beginner programmers to enable *progression* in programming.

References

Armoni, M., and Ben-Ari, M. (2010), Computer Science Concepts in Scratch, Licensed under Creative Commons. https://stwww1.weizmann.ac.il/scratch/scratch_en/.

Astrachan, O., and Wallingford, E. (1998), Loop Patterns. https://users.cs.duke.edu/~ola/patterns/plopd/loops.html.

Atkinson, R. K., Derry, S. J., Renkl, A., and Wortham, D. (2000), 'Learning from Examples: Instructional Principles from the Worked Examples Research', *Review of Educational Research*, 70: 181–214.

Aureliano, V. C. O., Tedesco, P. C. de A.R., and Caspersen, M. E. (2016), 'Learning Programming through Stepwise Self-Explanations', in *Proceedings of the 11th Conferencia Iberica de*, Sistemas y Technologias de Information.

Back, R. J. (1978), *On the Correctness of Refinement Steps in Program Development*, Helsinki, Finland: Department of Computer Science, University of Helsinki.

Back, R. J. (1998), *Refinement Calculus: A Systematic Introduction*, Berlin: Springer-Verlag.

Bennedsen, J., and Caspersen, M. E. (2004a). 'Programming in Context – A Model-First Approach to CS1', in *Proceedings of the Thirty-Fifth SIGCSE Technical Symposium on Computer Science Education, SIGCSE,* 477–81.

Bennedsen, J., and Caspersen, M. E. (2004b), 'Teaching Object-Oriented Programming – Towards Teaching a Systematic Programming Process', *in Proceedings of the Eighth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts*, 18th European Conference on Object-Oriented Programming.

Bennedsen, J., and Caspersen, M. E. (2005), 'Revealing the Programming Process', in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education, SIGCSE*, 186–90.

Bergin, J. (1999), Patterns for Selection. http://www.cs.uni.edu/~wallingf/patterns/elementary/papers/selection.pdf.

Brennan, K., Balch, C., and Chung, M. (2014), 'Creative Computing', Harvard Graduate School of Education. http://scratched.gse.harvard.edu/guide/.

Börstler, J., Caspersen, M. E., and Nordström, M. (2007), 'Beauty and the Beast – Toward a Measurement Framework for Example Program Quality'. Technical Report UMINF-07.23, Umeå, Sweden: Department of Computing Science, Umeå University.

Börstler, J., Caspersen, M. E., and Nordström, M. (2016), 'Beauty and the Beast: On the Readability of Object-Oriented Example Programs', *Software Quality Journal*, 24 (2), 231–46.

Börstler, J., Nordström, M., and Paterson, J. H. (2011), 'On the Quality of Examples in Introductory Java Textbooks', *Transactions on Computing Education*, 11: 1–21.

Caspersen, M. E. (2007), Educating Novices in the Skills of Programming, DAIMI PhD Dissertation PD-07-4, Denmark: Department of Computer Science, Aarhus University. http://www.cs.au.dk/~mec/dissertation/Dissertation.pdf.

Caspersen, M. E., and Bennedsen, J. (2007), 'Instructional Design of a Programming Course – A Learning Theoretic Approach', in *Proceedings of the 2007 International Workshop on Computing Education Research*, 111–22, Atlanta.

Caspersen, M. E., and Kölling, M. (2009), 'STREAM: A First Programming Process', *ACM Transactions on Computing Education*, 9 (1): 1–29.

Caspersen, M. E., and Nowack, P. (2013), 'Computational Thinking and Practice—A Generic Approach to Computing in Danish High Schools', in *Proceedings of the 15th Australasian Computer Education Conference*, 137–43, Adelaide, South Australia.

Chi, M. T. H., Bassok, M., Lewis, M., Reimann, M., and Glaser, R. (1989), 'Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems', *Cognitive Science*, 13: 145–82.

Chiu, J. L., and Chi, M. T. H. (2014), 'Supporting Self-Explanation in the Classroom', in A. Benassi, C. E. Overson and C. M. Hakala (eds), Applying Science of Learning in Education: Infusing Psychological Science into the Curriculum. http://teachpsych.org/ebooks/asle2014index.php.

Clark, R. C., Nguyen, F., and Sweller, J. (2005), *Efficiency in Learning: Evidence-Based Guidelines to Manage Cognitive Load*, Chichester: Wiley.

Christensen, H. B., and Caspersen, M. E. (2002), 'Frameworks in CS1 – A Different Way of Introducing Event-Driven Programming', in *Proceedings of the Seventh Annual Conference on Innovation and Technology in Computer Science Education*.

Clancy, M. (2004), 'Misconceptions and Attitudes That Interfere with Learning to Program', in S. Fincher and M. Petre (eds), *Computer Science Education Research*, 85–100, Abingdon: Taylor & Francis.

Dasgupta, S., and Resnick, M. (2014), 'Engaging Novices in Programming, Experimenting, and Learning with Data', *ACM Inroads*, 5 (4): 72–5.

Dijkstra, E. W. (1969), *Notes on Structured Programming*, TH Report 70. Technical University Eindhoven, Netherlands. https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF.

du Boulay, B. (1989), 'Some Difficulties of Learning to Program', *Studying the Novice Programmer*, 57–73, Hillsdale, NJ: Lawrence Erlbaum.

East, J. P., Thomas, S. R., Wallingford, E., Beck, W., and Drake, J. (1996), *Pattern-Based Programming Instruction*, in *Proceedings of the ASEE Annual Conference and Exposition*, Washington, DC.

Gries, D. (1974), 'What Should We Teach in an Introductory Programming Course?' *in Proceedings of the Fourth SIGCSE Technical Symposium on Computer Science Education*, 81–9.

Gries, D. (2002). 'Where Is Programming Methodology These Days?' *ACM SIGCSE Bulletin*, 34 (4), 5–7.

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

Knudsen, J. L., and Madsen, O. L. (1988), 'Teaching Object-Oriented Programming Is More Than Teaching Object-Oriented Programming Languages', *ECOOP '88 European Conference on Object-Oriented Programming*, 21–40. Berlin: Springer.

Kölling, M. (2003), The Curse of Hello World', Workshop on Learning and Teaching Object-Orientation—Scandinavian Perspectives, Oslo.

Kristensen, B. B., and Østerbye, K. (1994), 'Conceptual Modeling and Programming Languages', *ACM SIGPLAN Notices*, 29 (9): 81–90.

Lee, I., Martin, F., Denner, J., Coulter, B., Allan, W., Erickson, J., Malyn-Smith, J., and Werner, L. (2011), 'Computational Thinking for Youth in Practice', *ACM Inroads*, 2 (1): 32–7.

LeFevre, J., and Dixon, P. (1986), 'Do Written Instruction Need Examples?' *Cognition and Instruction*, 3: 1–30.

Madsen, O. L., Møller-Pedersen, B., and Nygaard, K. (1993), *Object-Oriented Programming in the BETA Programming Language,* Wokingham: Addison-Wesley.

Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2013), 'Learning Computer Science Concepts with Scratch', *Computer Science Education*, 23 (3): 239–64.

Meyer, B. (1993), 'Towards an Object-Oriented Curriculum', *Journal of Object-Oriented Programming*, 6 (2): 76–81.
Morgan, C. (1990), *Programming from Specifications*, Upper Saddle River, NJ: Prentice Hall.
Muller, O. (2005), 'Pattern Oriented Instruction and the Enhancement of Analogical Reasoning'. in *Proceedings of the 2005 International Workshop on Computing Education Research*, 57–67.
Pattis, R. E. (1990), 'A Philosophy and Example of CS-1 Programming Projects', in *Proceedings of the Twenty-First SIGCSE Technical Symposium on Computer Science Education*, 34–39.
Quilici, J. L., and Mayer, R. E. (1996), 'Role of Examples in How Students Learn to Categorize Statistics Word Problems', *Journal of Educational Psychology*, 88: 144–61.
Ragonis, N., and Ben-Ari, M. (2005), 'A Long-Term Investigation of the Comprehension of OOP Concepts by Novices', *Computer Science Education*, 15 (3): 203–21.
Resnick, M. (2014), 'Give P's a Chance', in *Constructionism and Creativity Conference*, Opening Keynote, Vienna.
Robins, A., Rountree, J., and Rountree, N. (2003), 'Learning and Teaching Programming: A Review and Discussion', *Journal of Computer Science Education*, 13 (2): 137–72.
Sajaniemi, J. (2008), 'Roles of Variables'. http://www.cs.joensuu.fi/~saja/var_roles/.
Schmolitzky, A. (2005), 'Towards Complexity Levels of Object Systems Used in Software Engineering Education', in *Proceedings of the Ninth Workshop on Pedagogies and Tools for the Teaching and Learning of Object-Oriented Concepts, 19th European Conference on Object-Oriented Programming*. https://tinyurl.com/yd9vv5w6.
Soloway, E. (1986), 'Learning to Program – Learning to Construct Mechanisms and Explanations', *Communications of the ACM*, 29 (9): 850–8.
Sowa, J. F. (1984). *Conceptual Structures: Information Processing in Mind and Machine,* Boston, MA: Addison-Wesley.
Spohrer, J. C., and Soloway, E. (1986), 'Novice Mistakes: Are the Folk Wisdoms Correct?', *Communications of the ACM*, 29 (7): 624–32.
Sweller, J., and Cooper, G. (1985), 'The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra', *Cognition and Instruction*, 2: 59–89.
Trafton, J. G., and Reiser, B. J. (1993), 'Studying Examples and Solving Problems: Contributions to Skill Acquisition', Washington, DC: Technical report, Naval HCI Research Lab.
VanLehn, K. (1996), 'Cognitive Skill Acquisition', *Annual Review of Psychology*, 47: 513–39.
Wirth, N. (1971), 'Program Development by Stepwise Refinement', *Communications of the ACM*, 14 (4): 221–7.

19

The Role of Design in Primary (K–5) Programming

Jane Waite

Chapter outline

Chapter synopsis

Teaching primary-aged children to program presents many challenges, including helping them to design, write and debug programs. Planning in K–5 literacy lessons and other subjects is a well-established activity in many classrooms, for example drawing a storyboard in literacy lessons to plan a story or drawing a circuit to be made in science. But in computing lessons, the phase of planning (or design) can be overlooked or even avoided.

Investigation of what design is going on in K–5 programming lessons and what concepts might be involved has attracted very little attention. Yet, in some computing curricula, teachers must support students to design and make programs. This chapter will look at studies in this area and reflect on how we might practically apply research findings in teaching practice.

19.1 Rationale for learning to design in K–5 programming

The study of learning how to design in programming has a long history for undergraduate learners and those in industry. Soloway proposed that teaching programming was not just about the syntax and semantics of a programming language. Instead, it was about teaching problem-solving and that students should be explicitly taught that programming is a design discipline (Soloway, 1986: 853). Denning (2011: 88) similarly focused his attention on the importance of design as he categorized design as one of the great principles of computing.

Despite these clear recommendations to teach how to design programs, in K–12 contexts program design has until recently been generally lacking both in research undertaken (Rich, Strickland and Franklin, 2017; Oleson, Wortzman and Ko, 2021) and in the development of teaching and learning resources (Falkner and Vivian, 2015).

Sometimes design is mentioned in passing as part of curriculum research activities. More recently, perhaps because of the calls for design research, pockets of design-focused research have started to appear. For example, Grover, Pea and Cooper (2015) have designed a blended computer science course for middle school learners that integrates design. Similarly, but for younger learners, Hansen et al. (2016) have developed a K–5 programming course with elements of design. Physical computing research sometimes includes design activities, including research in kindergarten (Sullivan and Bers, 2018), middle school and high school contexts (e.g. Kafai, Fields and Searle, 2012; Przybylla and Romeike, 2017).

With respect to design-specific research, Schulte et al. (2017) have proposed a theoretical model for incorporating design in school programming lessons that includes two interleaved learning paths of exploration and design. Industry approaches have been tailored for the classroom, such as the engineering design process in early primary years (Bers, 2017) and agile methods for middle and high school (Kastl and Romeike, 2015). Loksa et al. (2016) have also investigated emphasizing self-awareness of the design process in high school students.

Over the past five to six years, a set of studies have been conducted explicitly looking at design in K–5 programming classrooms in England (Waite et al., 2018a, 2018b, 2020;Waite, 2022); this set of studies forms the basis for the design concepts discussed in this chapter.

Despite this promising start, the work about design in programming in schools is limited in scale and the length of study, and there are few studies in the K–5 context. Yet, K–5 teachers worldwide are being increasingly required to teach programming in their classrooms and within this, to some degree, design. Programming in K–8 has been introduced to curricula in Finland, Australia, Croatia, Slovenia, Ukraine, Greece, South Korea and some parts of Spain, Italy, Hong Kong, Germany, the United States and the UK (Rich et al., 2018), and the list is growing.

Examples of design in K–5 curriculum

How design is explicitly incorporated in each country's curriculum varies; we take just two curricula examples, one from England and one from the United States.

In England, educators have been required since 2014 to teach grade K–1 students to:

- 'Understand what algorithms are, how they are implemented as programs on digital devices, and that programs execute by following precise and unambiguous instructions
- Create and debug simple programs'. (Department for Education, 2013: 2)

In the United States, the CSTA framework for K–1 children is similar. Students should:

- 'Model daily processes by creating and following algorithms (sets of step-by-step instructions) to complete tasks. (1A-AP-08)
- Decompose (break down) the steps needed to solve a problem into a precise sequence of instructions. (1 A-AP-11)
- Develop plans that describe a program's sequence of events, goals, and expected outcomes (1A-AP-11)'. (CSTA, 2017)

For slightly older learners in England, grade 2–5 learners should:

- 'Design, write and debug programs that accomplish specific goals, including controlling or simulating physical systems; solve problems by decomposing them into smaller parts
- Use sequence, selection, and repetition in programs; work with variables and various forms of input and output'. (Department for Education, 2013: 2)

In the United States, the CSTA curriculum for grade 2–5 requires teachers to enable students to:

- 'Compare and refine multiple algorithms for the same task and determine which is the most appropriate (1B-AP-08)
- Decompose (break down) problems into smaller, manageable subproblems to facilitate the program development process (1B-AP-11)
- Use an iterative process to plan the development of a program by including others' perspectives and considering user preferences (1B-AP-13)'. (CSTA, 2017)

Whether all teachers implement these requirements in the same way is an open question. To start reflecting on this, we next look at research investigating K–5 teachers' experiences of design (or planning) in computing and other subjects.

19.2 Teaching K–5 planning in other subjects

K–5 teachers are likely to be generalists who teach their young students all subjects, from literacy to science. Teachers, therefore, have the experience of using design, sometimes called planning, in the teaching of other subjects.

Here, when we talk about K–5 planning, we refer to deciding what will be included in our finished artefact and figuring out how the components of that artefact will fit together. For example, if teaching seven-year-old pupils how to write a story, the teacher might discuss and draw the setting, including the characters engaged in the story and sketch out a storyboard. This planning activity is not saying who will do the work of translating the plan into a finished piece of writing; it is the design that outlines what will be written.

In the research of teaching writing in literacy lessons, investigation of the planning aspect of the process for writing has a long, rich history and is recommended to K–5 teachers (Schunk and Swartz, 1993; Graham et al., 2012; Higgins et al., 2021). Planning has been identified as an essential metacognitive or executive process that supports self-regulation in writing (Berninger et al., 2002). Planning separates the task of gathering ideas and developing an initial overall piece from the job of then implementing this as the written text. Learners can annotate their plan with useful words and phrases without contending with how the fragments might be joined to create larger structures. At the end of writing, the plan can be used to check that all intended parts are included, checking for completeness and cohesion.

Similarly, in design and technology lessons, the phase of planning out, designing, what is to be made is a familiar phase in some K–5 classrooms and may be done by drawing the intended product. In mathematics teaching, planning how to tackle different types of problems is likely to be taught, as is thinking about how to conduct a science experiment, such as drawing a circuit to be built. Whether these planning activities are formally documented in specific formats will vary by pedagogical approach and classroom context.

19.3 In classroom use of, and difficulties with, teaching K–5 program design

To discover what K–5 teachers think about design in programming and what they are doing in lessons, Waite et al. (2018a) surveyed this topic with primary school teachers working in England.

The researchers asked teachers about their views of the usefulness of design in programming lessons and their views of the usefulness of planning in writing lessons. Of the 207 respondents, 82 per cent of teachers reported that design was very useful or essential (see Table 19.1), and 78 per cent thought the same for the usefulness of planning in teaching writing in literacy lessons.

Table 19.1 Usefulness of design and usefulness of planning

Usefulness	Design (%)	Cumulative (%)	Planning (%)	Cumulative (%)
Essential	53% (110)	**(53%)**	51% (105)	**(51%)**
Very useful	29% (59)	**(82%)**	27% (56)	**(78%)**
Somewhat useful	15% (32)	**(97%)**	16% (33)	**(94%)**
A little useful	3% (6)	**(100%)**	4% (9)	**(98%)**
Not useful	0% (0)	**(100%)**	2% (4)	**(100%)**

Source: Waite et al. (2018a: 4).

Table 19.2 Use of design and use of planning

Use	Design (%)	Cumulative (%)	Planning (%)	Cumulative (%)
Always	20% (42)	**(20%)**	57% (58)	**(57%)**
Usually	24% (49)	**(44%)**	31% (31)	**(88%)**
Sometimes	36% (75)	**(80%)**	9% (9)	**(97%)**
Rarely	13% (26)	**(93%)**	2% (2)	**(99%)**
Never	7% (15)	**(100%)**	1% (1)	**(100%)**

Source: Waite et al. (2018a: 5).

Teachers were also asked about their use of design in programming lessons and their use of planning in writing lessons; here there was a marked difference compared to views on usefulness. Only 44 per cent of teachers said they usually or always used designs in programming lessons compared to 88 per cent saying they usually or always used planning in writing lessons (see Table 19.2).

Teachers who have started to use design in programming lessons have reported using design for a wide range of purposes, including managing the process of program development, for assessment and differentiation of lessons (Waite et al., 2018a,_b, 2020).

Example: Uses of design

Students can use design

- To gather ideas
- To check the design with the intended user
- To distinguish between the different parts of the design (design components)
- To separate the effort of idea generation and implementation
- To self-assess confidence to be able to implement the different aspects of the design
- To annotate with code snippets
- As an aide memoire to tick off what they have done and need to do next
- As a contract with a partner for pair programming
- To self-assess against design criteria
- To help with debugging

Teachers can use design

- To reveal students' ideas and understanding
- To help work out what to teach next
- For formative assessment
- For summative assessment
- To reveal and explain the value of design
- To differentiate and scaffold teaching

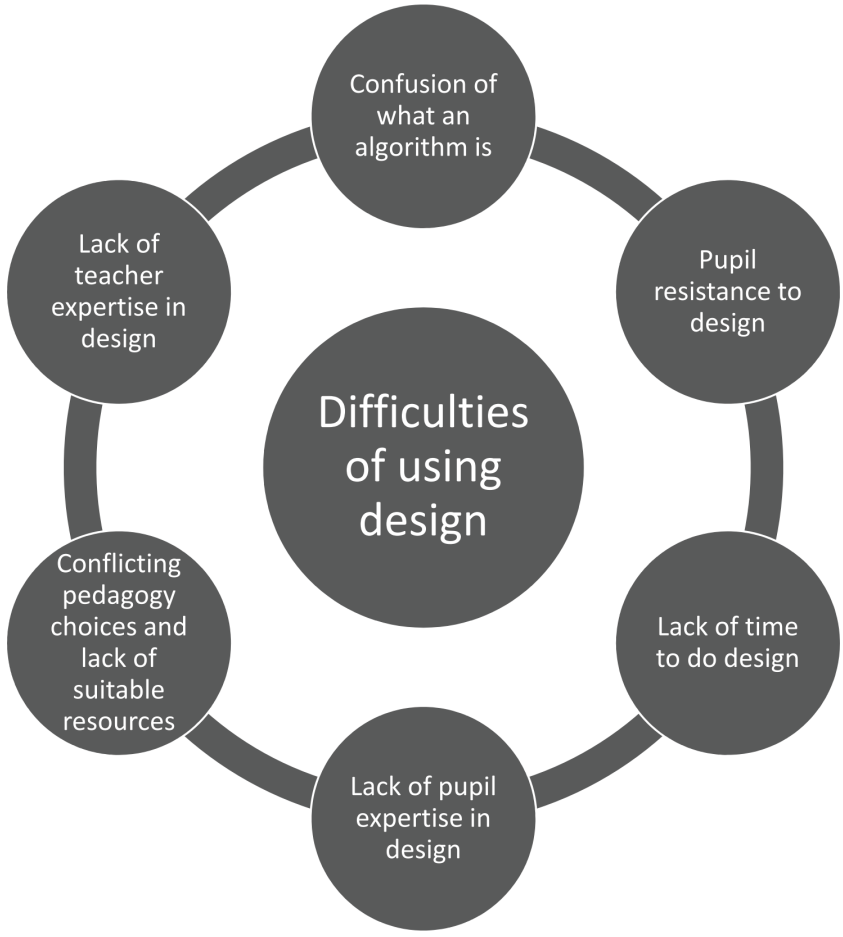

Figure 19.1 Difficulties of using design in primary programming classrooms (*Source:* Waite et al., 2020: 26).

However, not all teachers use design. What stops some K–5 teachers from incorporating design in programming lessons has been reported to include six difficulties for using design in K–5 programming classrooms (see Figure 19.1) (Waite et al., 2020).

Waite et al. (2020) found their surveyed K–5 teachers reported a significant reason for not doing design is pupil resistance. Teachers said pupils want to get on and code or think design is boring, as one teacher's comment exemplified: 'A minority find it useful but I think most find it unnecessary and a chore. They don't link it to computing' (English primary teacher).

This example leads to the theme of a lack of pupil expertise. Teachers reported that students lacked the knowledge and experience of program design. Teachers mentioned that students created designs that were too ambitious or that students had difficulties from sticking too closely to designs.

A further major difficulty highlighted was that there was not enough time to do design in programming lessons.

Teachers also reported they lacked expertise and confidence in teaching design. An issue linked to expertise was that there was no consensus from the surveyed teachers about what an algorithm was and how this might relate to the design of a programming project.

Another theme was the lack of resources with design in them or conflicting pedagogies. This latter aspect of pedagogical conflicts is exemplified by one teacher's comments on some students preferring an organized approach to design versus others preferring an exploratory approach to design.

> Some like the organisation and ordered approach [to design], but some prefer to jump in, try out ideas, select what works, then plan. It does help them think about why they are programming and what they want to happen, especially thinking about how they order and sequence. (English primary teacher)

19.4 K–5 design concepts and opportunities for classroom use

To investigate how difficulties with K–5 design in programming projects might be addressed, a design toolkit of concepts involved in K–5 design has been co-designed with teachers. Twenty-eight expert primary teachers who reported using design in programming lessons have worked on the toolkit. The design toolkit includes design concepts that may provide a way to help to address the difficulty of teachers' lack of expertise about K–5 design and work towards remedying the confusion over what an algorithm is. By having a better understanding of design, teachers may decide that design is worth the time needed and may start to build techniques to overcome student resistance and address students' low expertise levels. But more research is needed to evaluate the impact of design on student outcomes.

We highlight six of the eleven design concepts from the toolkit here, as shown in Figure 19.2:

- Activity genre
- Common design patterns
- Design component
- Design format
- Design approach
- Levels of abstraction

Activity genre and simple K–5 design patterns

Concept descriptions and examples

Activity genre is the broad type of programming project. For example, K–5 project genres include route-based, animations, quizzes, games and simulations, physical computing and tools or aids. Some projects will be of more than one genre; for example, children might use a microcontroller to build a toy that navigates a maze – this would be a route-based physical computing genre.

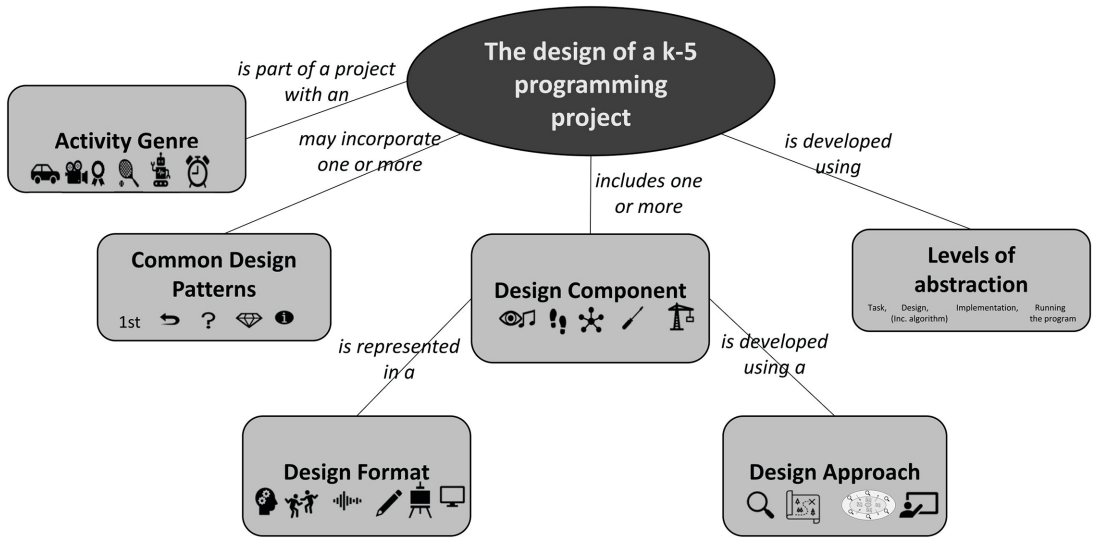

Figure 19.2 Design concepts for K–5 programming projects (*Source:* Waite, 2022).

A simple K–5 design pattern is a useful functionality that is commonly used in K–5 projects. For example, managing scores might be common in K–5 games and quizzes, and asking for a username to personalize user output might be useful in animations, quizzes, games and simulations.

About the concept

Genre is a useful way of grouping types of writing, films and computer games. In research on teaching literacy, genre, such as poetry, recounts and traditional stories are used to help students learn the common features of written pieces (Rose, 2009). Similarly, types of programming projects such as animations, quizzes, games and simulations are noted in research and lesson planning (Brennan and Resnick, 2012; Spieler et al., 2017).

Some research suggests design patterns are associated with specific genres. For example, Basawapatna et al. (2011) identified collision, generation and absorption as common design patterns for games and simulation genres. Design patterns require particular programming constructs to be used when they are implemented. For example, games are likely to have design patterns that keep scores and manage lives, which require variables. Similarly, games are likely to incorporate user input that controls the action, requiring the use of if-then-else statements (conditionals). In contrast, there are limited opportunities to use variables and conditionals in simple animations. Therefore, some programming project genres may limit or extend student learning by their common design patterns. At the same time, some genres appear to be more or less appealing to some groups of children. For example, girls have been found to prefer to create animations and boys to create games, in which case, genre and student preference may disadvantage some students to learn and make progress (Troiano et al., 2020).

Example: Classroom opportunities

Check if children have the opportunity to work with a range of genres.

- Ensure all children can work with a range of K–5 design patterns and associated programming constructs. For example, if making animations, ask students to add interactivity, asking for a user's nickname (to be stored in a variable for later use) and user choices (requiring if-then-else conditional statements).
- Teach children about genre and common design patterns so they can reuse the knowledge across projects.

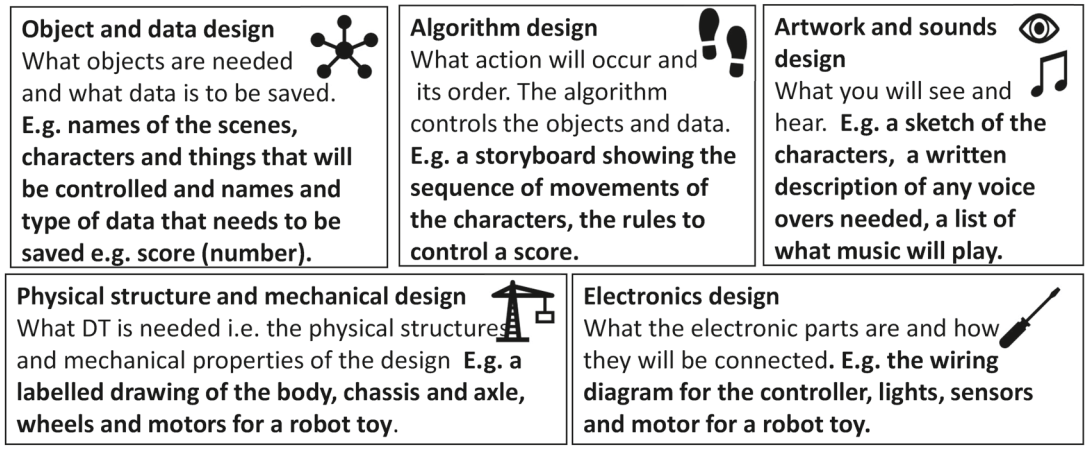

Figure 19.3 Design components of K–5 programming projects (*Source:* Waite, 2022).

Design components

Concept descriptions and examples

To help young learners understand more about the different aspects of design, the design components concept has been suggested (Waite, 2022). When designing a project, several design components are developed (see Figure 19.3). For most K–5 projects, there will be design components of an object and data design, an algorithm design and an artwork and sound design. For some projects, there may also be a physical structure and mechanical design or an electronics design.

About the concept

K–5 programming projects are expected to have an algorithm design as the code is an implementation of an algorithm, and programming projects usually require students to write code. When learners

design their algorithms, they break the action down into parts and decide upon the order in which these parts will occur. As well as an algorithm, K–5 designs will usually have an object and a data design. This is to help learners start to think about what thing(s) will be controlled in some way by the algorithm and to name them. For example, in an animation, a character or scene (an object) will change its appearance. In a route-based activity, a programmable toy or on-screen character (an object) will be programmed to navigate a route. It is hoped that getting students to think about design components may help them as they implement their programs, but more research is needed in this area to confirm this concept.

A little more about algorithms

An algorithm, in the computer science sense, is unambiguous; 'algorithms are directions to control a computational model (abstract machine) to perform a task' (Denning, 2017: 4), and each operation has a 'well defined effect that can be carried out by a machine' (Denning, 2017: 8).

Waite et al. (2020) have argued that in K–5 contexts, students often do now know what is practicable at the start of their programming projects and gradually refine their ideas about their design, including the algorithm, of what should happen as they go along. Also, in K–5 contexts, the formats used by students to represent design, such as drawings or physical enactments, do not foster precise definitions. Therefore, it is likely their representations of algorithms will be ambiguous. Teachers must therefore consider whether they call these imprecise representations an algorithm. Further research is needed to investigate what impacts this ambiguous view will have as students learn more about the precision required of an algorithm (Waite et al., 2020).

Example: Classroom opportunities

Help students to become aware of the different design components. For example, Figure 19.4 shows a design in which the characters (objects) are named in a list, the algorithm is implied through the numbered speech bubbles and the artwork is the images of the characters and scene.

Design formats

Concept descriptions and examples

The design format is the media (or representation) in which the design exists. The format could be written notes, a drawn labelled diagram or a storyboard (see Figure 19.5). For a single component, such as the algorithm design, there may be more than one format for a project. For example, the algorithm may be physically enacted, verbally explained and indicated on a storyboard through numbered events. A single design format may include representations of more than one design component. For example, a drawn storyboard might include the artwork, algorithm, object and data design (see Figure 19.4).

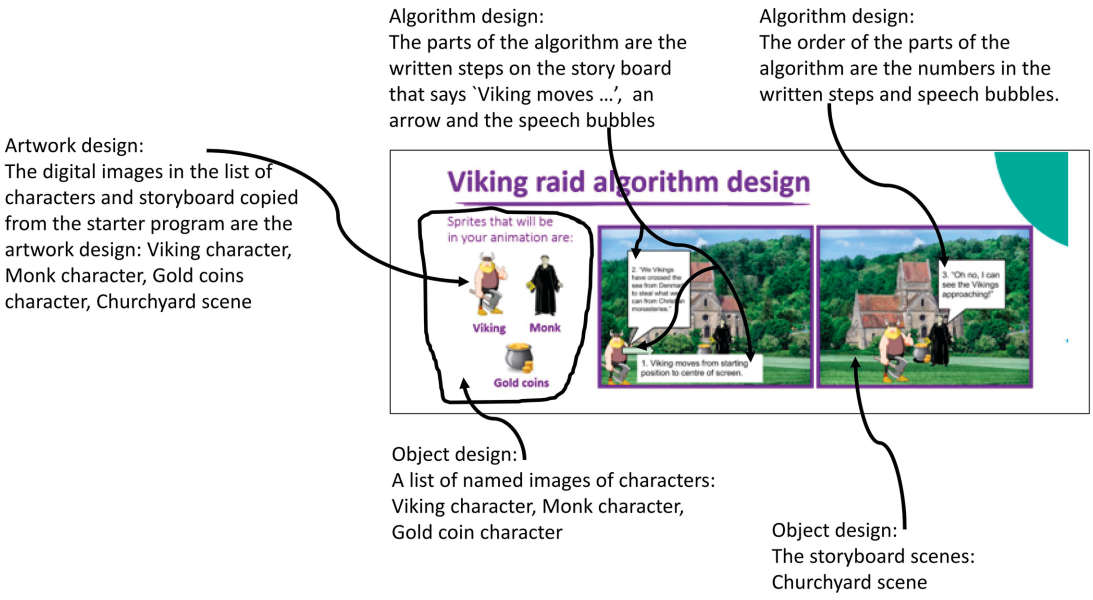

Figure 19.4 Example design including annotations of the design components (*Source:* original design from Barefoot Computing Resources).

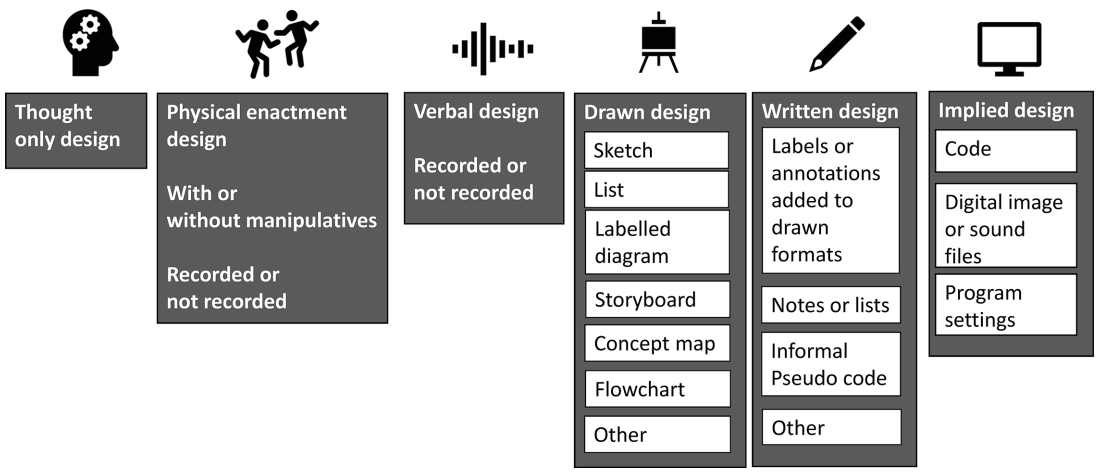

Figure 19.5 Example design formats for K–5 programming projects (*Source:* Waite, 2022).

About the concept

Teachers are likely to build a repertoire of design formats that they see as being useful for the projects that they use in programming lessons. Some teaching resources may suggest specific formats, such as walking out a route for a programmable toy activity (a physical enactment design). Sometimes teachers may use familiar formats from learning in other subjects.

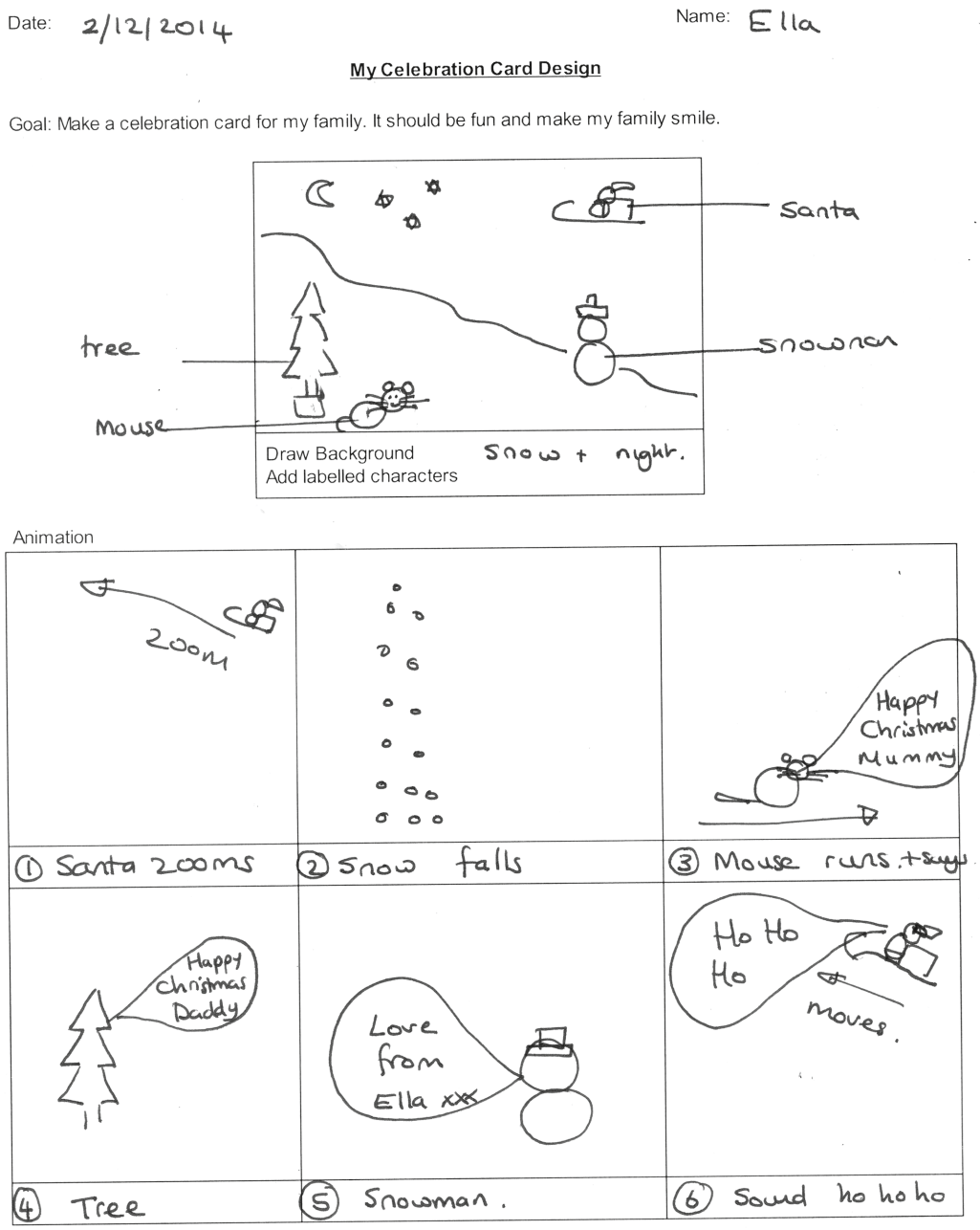

Figure 19.6 An example design with a labelled diagram and storyboard.

Figure 19.6 is an example of a K–5 design for a Christmas card animation. The design includes two formats, a labelled diagram and a storyboard. In the labelled diagram, the characters and background (objects) needed for the project are drawn and labelled. The algorithm design is

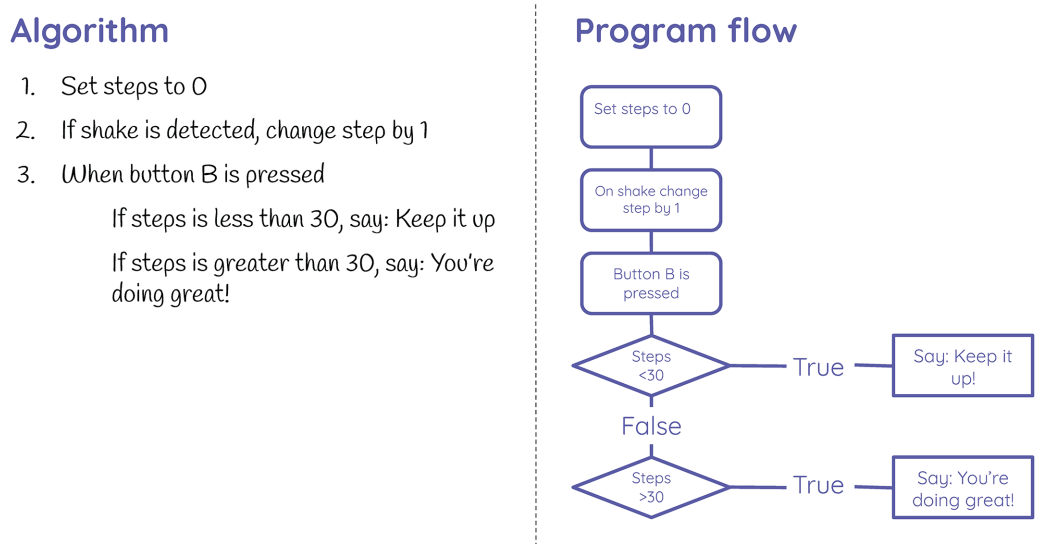

Figure 19.7 An example design with a flowchart and written algorithm.

shown as the sequence of events in the storyboard, including the numbering and arrows that show movement. In the second box, the student indicates they want to show snow falling. As they start their design, they will need to work out how to implement this. It may be that they use a series of backgrounds or that they program individual snowflakes. The algorithm of each of these options is very different, but it is unlikely that young students will represent these algorithms formally.

Whether designs are updated as the development progresses or how precise student's designs are will vary by student and teacher. For example, as shown in Figure 19.7, some resources suggest using a flowchart with older K–5 learners to create more precise designs using formal notation.

More research is needed to investigate what level of precision is most useful for teaching and learning and, similarly, whether young learners should call an imprecise description of events an algorithm.

Example: Classroom opportunities

- Make students aware of verbal or acted-out designs so that they realize they are designing their projects.
- Provide students with format templates.
- Model how to create storyboards and other design formats.

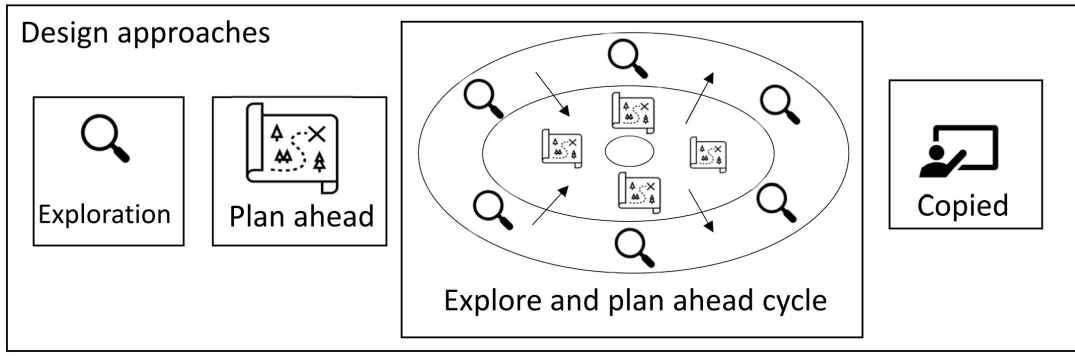

Figure 19.8 Design approaches for K–5 projects (*Source:* Waite, 2022).

Design approaches

Concept descriptions and examples

Design approaches are the approaches that students follow to develop their design. An example approach is exploratory, where students do not plan very far ahead, trying out ideas as they go along. Another approach is plan-ahead, where students think about the main characters and events in the product to be made before they start to implement their ideas. As shown in Figure 19.8, students are likely to combine an exploratory and plan-ahead approach. If students follow instructions to create a product originally designed by someone else, they will be using a copied design approach.

About the concept

The design approach sits within an overall project development process. For example, Bers (2017) has suggested a project development approach for very young learners to follow of a simplified engineering design process (see Figure 19.9). This process has been successfully used with kindergarten and K–1 learners and includes a defined phase where they plan their project. What design approach is used within this planning phase is not stipulated.

Beginner programmers are not like expert programmers; beginners are learning what a programming language's commands will do and what is possible as they build a project; they are constructing their mental models of programming languages as they go along.

Therefore, it is not likely that young students will be able to formulate a precise design that they can successfully implement as they embark on their project. Schulte et al. (2017) consider this in their theoretical design exploration model. In this model, students are expected to continually move between two learning paths, one of exploration and learning about what can be designed and one of design where they use this knowledge to think ahead and plan what to do.

The exploration approach can be linked to the concept of bricolage. Bricolage is an informal, artisan approach where the craftworker discovers the properties of the materials they work with and shapes a product in an experimental way (Levi-Strauss, 1968). This approach is often associated with Papert's constructionism, and Turkle and Papert (1991) proposed that children were either bricoleurs or planners.

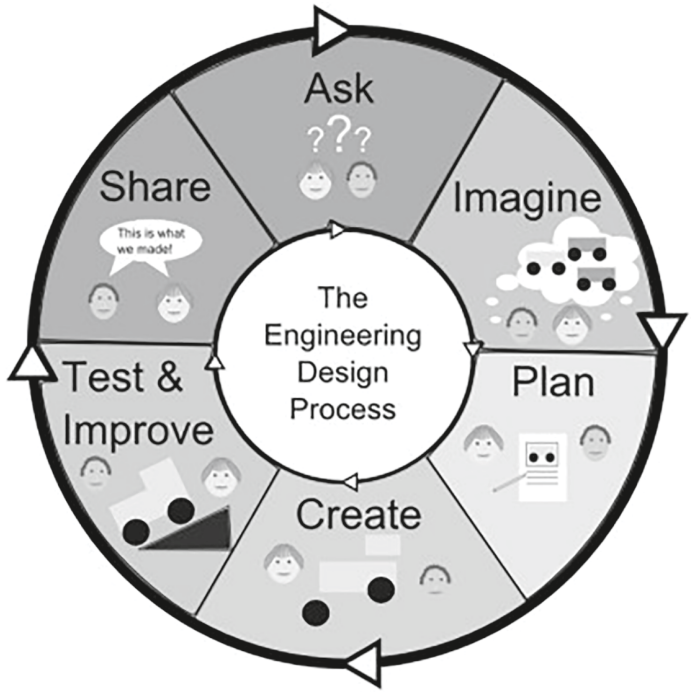

Figure 19.9 The simplified engineering design process used with K–2 children (*Source:* Bers et al., 2014: 155).

Example: Classroom opportunities

- If your students copy designs, create a plan-ahead design and show them the parts they are implementing.
- If your students tend not to think ahead, challenge them to explain what they want to achieve.
- If your students get bogged down in a plan-ahead format and are nervous about exploring, model how to explore and exemplify how to use the new-found functionality.

Levels of abstraction

Concept descriptions and examples

When we create a program, there are four 'levels' we can consider. These can be called levels of abstraction as they are different ways of thinking about a programming activity. As shown in Figure 19.10, levels of task, design, implementation and running the program have been suggested

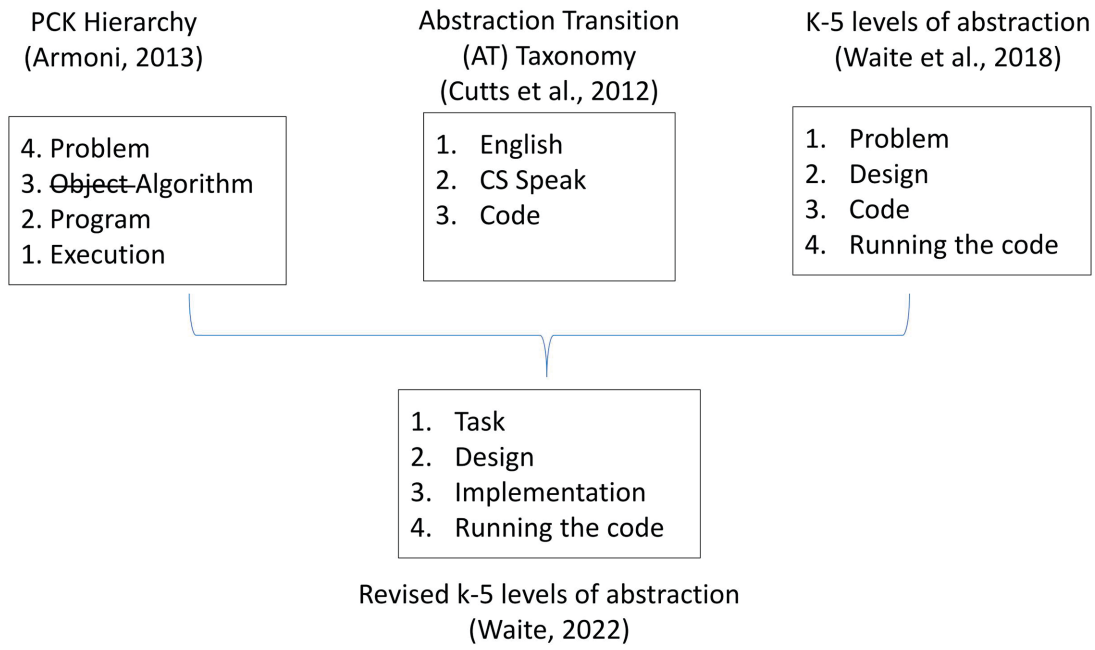

Figure 19.10 Revised K–5 levels of abstraction for programming projects and the models that have informed the new version (*Source:* Waite, 2022).

in a revised K–5 programming levels of abstraction model. The levels are not a linear set of stages nor strict steps to complete a programming project; they are a set of views of a project. Not all programming activities will incorporate all levels. The order in which pupils work on levels will vary for different activities. Also, it is likely we won't work on each level just once. Instead, we will move between the levels many times and start at different levels.

About the concept

The K–5 levels of abstraction model builds on the work of researchers who have studied students learning to program (see Figure 19.10). Perrenet, Groote and Kaasenbrood (2005) suggested a levels of abstraction model for undergraduates. Armoni (2013) adapted this model by renaming the object level to the algorithm level. These models have been combined using feedback from K–5 students and their teachers, and levels have been renamed for primary school use (Waite et al., 2018a; Waite, 2022).

The model suggested by Armoni has been used in high school research and has been found to improve student programming outcomes by supporting students to understand the level they are working at, including careful use of level-specific vocabulary (Statter and Armoni, 2016). Of note, girls made more progress than boys when using the framework, and the authors suggest that the model may increase girls' self-efficacy about learning to program (Statter and Armoni, 2017).

Task: The task is a summary written or verbal description of a program.

Design: The design represents the algorithm and other design components.

Implementation: The implementation level is the code itself, any files created for artwork or sounds, any program setting that has created objects such as sprites and variables and any physical structures and wiring for physical computing activities.

Running the program: This is either the code running or any reference to the program's output, such as when debugging the code.

Example: Classroom opportunities

- Review your classroom use of programming project vocabulary and be more level-specific.
- Explain the levels to students and ask them to spot when they and their peers move between levels.
- Model techniques to help students move between levels, such as adding code snippets annotations to the design.

19.5 Conclusion

Design is important in the teaching of programming to older learners and is often included in K–5 curricula but seems to be overlooked. Teachers face various challenges in finding resources, gaining experience and overcoming student resistance to using design. Emerging research is starting to suggest concepts that might help teachers and learners overcome difficulties with design. However, research is needed to evidence how and what design should be introduced in class and how to support teachers.

Key points

- Design is an essential aspect of program development and is often included in K–5 curricula, but what this means in classroom practice is not agreed.
- K–5 students and their teachers are likely to do planning, a form of design, in other subjects, such as storyboarding in literacy lessons; therefore, cross-curricular planning expertise, experience and expectations can be built upon in programming lessons.
- K–5 teachers may have difficulties implementing design in their programming lessons, including student resistance, a lack of their own and their students' experience and understanding of design, a lack of time to do design, conflicting views of how to do it and a lack of resources that include design.

- Some K–5 programming teachers are starting to overcome difficulties in including design in their teaching and use techniques such as modelling how to do design, using design templates, asking students to annotate designs with useful code snippets and highlighting the value of design.
- Having a set of K–5 concepts may help teachers and students introduce design in classrooms; concepts will provide a common vocabulary and improve knowledge about design and increase expertise in classroom programming design.

For further reflection

- What are your experiences of planning in other subjects (e.g. in teaching literacy)? How have these experiences shaped your view of planning and design?
- Do you agree with the Turkle and Papert (1991) view that pupils are either bricoleurs or planners? What are the benefits and disadvantages of design?
- Which of the design concepts, activity genre, common design patterns, design components, design formats, design approaches or levels of abstraction might be most useful for teachers and students? How and when might they be incorporated into classroom practice?

References

Armoni, M. (2013), 'On Teaching Abstraction in Computer Science to Novices', *Journal of Computers in Mathematics and Science Teaching*, 32, 265–84.

Basawapatna, A., Koh, K. H., Repenning, A., Webb, D. C., and Marshall, K. S. (2011), 'Recognizing Computational Thinking Patterns', in *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, 245–50.

Berninger, V. W., Vaughan, K., Abbott, R. D., Begay, K., Coleman, K. B., Curtin, G., Hawkins, J. M., and Graham, S. (2002), 'Teaching Spelling and Composition Alone and Together: Implications for the Simple View of Writing', *Journal of Educational Psychology*, 94 (2), 291–304. https://doi.org/10.1037/0022-0663.94.2.291.

Bers, M. U. (2017), *Coding as a Playground: Programming and Computational Thinking in the Early Childhood Classroom*. New York: Routledge.

Bers, M. U., Flannery, L., Kazakoff, E. R., and Sullivan, A. (2014), 'Computational Thinking and Tinkering: Exploration of an Early Childhood Robotics Curriculum', *Computers & Education*, 72, 145–57. https://doi.org/10.1016/j.compedu.2013.10.020.

Brennan, K., and Resnick, M. (2012), 'New Frameworks for Studying and Assessing the Development of Computational Thinking', in *Proceedings of the 2012 Annual Meeting of the American Educational Research Association, Vol. 1*, 25, Vancouver.

Computer Science Teachers Association (2017), CSTA K-12 Computer Science Standards, Revised 2017. http://www.csteachers.org/standards.

Cutts, Q., Esper, S., Fecho, M., Foster, S. R., and Simon, B. (2012), 'The Abstraction Transition Taxonomy: Developing Desired Learning Outcomes through the Lens of Situated Cognition', in *Proceedings of the Ninth Annual International Conference on International Computing Education Research - ICER '12*, 63. https://doi.org/10.1145/2361276.2361290

Denning, P. J. (2011), 'The Great Principles of Computing', in M. Pitici (ed.), *The Best Writing on Mathematics 2011,* 82–92, Princeton, NJ: Princeton University Press. https://doi.org/10.1515/9781400839544.82

Denning, P. J. (2017), 'Remaining Trouble Spots with Computational Thinking', *Communications of the ACM*, 60 (6), 33–9. https://doi.org/10.1145/2998438.

Department for Education, England (2013), Computing Programmes of Study Key Stages 1 and 2 National Curriculum in England. https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study.

Falkner, K., and Vivian, R. (2015), 'A Review of Computer Science Resources for Learning and Teaching with K–12 Computing Curricula: An Australian Case Study', *Computer Science Education*, 25 (4), 390–429. https://doi.org/10.1080/08993408.2016.1140410.

Graham, S, Bolinger, A., Olson, C., D'Aoust, C., MacArthur, C., McCutchen, D., and Olinghouse, N. (2012), *Teaching Elementary School Students to Be Effective Writers*. What Works Clearinghouse, National Center for Education Evaluation and Regional Assistance, Institute of Education Sciences, U.S. Department of Education, 109. https://files.eric.ed.gov/fulltext/ED533112.pdf

Grover, S., Pea, R., and Cooper, S. (2015), 'Designing for Deeper Learning in a Blended Computer Science Course for Middle School Students', *Computer Science Education*, 25 (2), 199–237. https://doi.org/10.1080/08993408.2015.1033142.

Hansen, A. K., Hansen, E. R., Dwyer, H. A., Harlow, D. B., and Franklin, D. (2016), 'Differentiating for Diversity: Using Universal Design for Learning in Elementary Computer Science Education', in *Proceedings of the 47th ACM Technical Symposium on Computing Science Education–SIGCSE '16*, 376–81. https://doi.org/10.1145/2839509.2844570.

Higgins, S., Martell, T., Waugh, D., Henderson, P., Sharples, J., Bilton, C., and Duff, A. (2021), *Improving Literacy in Key Stage 2 Guidance Report*. Education Endowment Fund. https://educationendowmentfoundation.org.uk/education-evidence/guidance-reports/literacy-ks2.

Kafai, Y. B., Fields, D. A., and Searle, K. A. (2012), Making Technology Visible: Connecting the Learning of Crafts, Circuitry and Coding in Youth e-Textile Designs, in *The Future of Learning: Proceedings of the 10th International Conference of the Learning Sciences* (ICLS 2012), vol. 1, 8.

Kastl, P., and Romeike, R. (2015), ' "Now They Just Start Working, and Organize Themselves" First Results of Introducing Agile Practices in Lessons', in *Proceedings of the Workshop in Primary and Secondary Computing Education*, 25–8. https://doi.org/10.1145/2818314.2818336.

Levi-Strauss, C. (1968), *The Savage Mind*, Chicago, IL: University of Chicago Press.

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., and Burnett, M. M. (2016), 'Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance', in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 1449–61. https://doi.org/10.1145/2858036.2858252.

Oleson, A., Wortzman, B., and Ko, A. J. (2021), 'On the Role of Design in K–12 Computing Education', *ACM Transactions on Computing Education*, 21 (1), 1–34. https://doi.org/10.1145/3427594.

Perrenet, J., Groote, J. F., and Kaasenbrood, E. (2005), 'Exploring Students' Understanding of the Concept of Algorithm: Levels of Abstraction', in *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education - ITiCSE '05*, 64. https://doi.org/10.1145/1067445.1067467.

Przybylla, M., and Romeike, R. (2017), 'The Nature of Physical Computing in Schools: Findings from Three Years of Practical Experience', in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, 98–107. https://doi.org/10.1145/3141880.3141889.

Rich, K., Strickland, C., and Franklin, D. (2017), 'A Literature Review through the Lens of Computer Science Learning Goals Theorized and Explored in Research', in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 495–500. https://doi.org/10.1145/3017680.3017772.

Rich, P. J., Browning, S. F., Perkins, M., Shoop, T., Yoshikawa, E., and Belikov, O. M. (2018), 'Coding in K–8: International Trends in Teaching Elementary/Primary Computing', *TechTrends*, 63 (3), 311–29. https://doi.org/10.1007/s11528-018-0295-4.

Rose, D. (2009), 'Writing as Linguistic Mastery: The Development of Genre-Based Literacy Pedagogy', in *The SAGE Handbook of Writing Development*, 151–66. Thousand Oaks, CA: Sage. https://doi.org/10.4135/9780857021069.n11.

Schulte, C., Magenheim, J., Muller, K., and Budde, L. (2017), 'The Design and Exploration Cycle as Research and Development Framework in Computing Education', in *2017 IEEE Global Engineering Education Conference (EDUCON)*, 867–76. https://doi.org/10.1109/EDUCON.2017.7942950.

Schunk, D. H., and Swartz, C. W. (1993), 'Goals and Progress Feedback: Effects on Self-Efficacy and Writing Achievement', *Contemporary Educational Psychology*, 18 (3), 337–54. https://doi.org/10.1006/ceps.1993.1024.

Soloway, E. (1986), 'Learning to Program—Learning to Construct Mechanisms and Explanations', *Communications of the ACM*, 29 (9), 850–8. https://doi.org/10.1145/6592.6594.

Spieler, B., Schindler, C., Slany, W., Mashkina, O., Beltrán, M. E., and Brown, D. (2017), *Evaluation of Game Templates to support Programming Activities in Schools*. 10. https://arxiv.org/abs/1805.04517.

Statter, D., and Armoni, M. (2016), 'Teaching Abstract Thinking in Introduction to Computer Science for 7th Graders', in *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*, 80–3. https://doi.org/10.1145/2978249.2978261.

Statter, D., and Armoni, M. (2017), 'Learning Abstraction in Computer Science: A Gender Perspective', in *Proceedings of the 12th Workshop on Primary and Secondary Computing Education*, 5–14. https://doi.org/10.1145/3137065.3137081.

Sullivan, A., and Bers, M. U. (2018). 'Dancing Robots: Integrating Art, Music, and Robotics in Singapore's Early Childhood Centers', *International Journal of Technology and Design Education*, 28 (2), 325–46. https://doi.org/10.1007/s10798-017-9397-0.

Troiano, G. M., Chen, Q., Alba, Á. V., Robles, G., Smith, G., Cassidy, M., Tucker-Raymond, E., Puttick, G., and Harteveld, C. (2020), 'Exploring How Game Genre in Student-Designed Games Influences Computational Thinking Development', in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 1–17. https://doi.org/10.1145/3313831.3376755.

Turkle, S., and Papert, S. (1991), 'Epistemological Pluralism and the revaluation of the concrete', in S. Papert and I. Harel eds, *Constructionism: Research Reports and Essays 1985–90*, 30, Cambridge, MA: MIT Media Laboratory.

Waite, J. (2022), Design in Primary Programming. PhD Thesis in Progress. Queen Mary University of London.

Waite, J., Curzon, P., Marsh, W., and Sentance, S. (2018a), 'Comparing K–5 Teachers' Reported Use of Design in Teaching Programming and Planning in Teaching Writing', in *Proceedings of the 13th Workshop in Primary and Secondary Computing Education*, 1–10. https://doi.org/10.1145/3265757.3265761.

Waite, J., Curzon, P., Marsh, W., Sentance, S., and Hadwen-Bennett, A. (2018b), 'Abstraction in Action: K–5 Teachers' Uses of Levels of Abstraction, Particularly the Design Level, in Teaching Programming', *International Journal of Computer Science Education in Schools*, 2 (1), 14–40. https://doi.org/10.21585/ijcses.v2i1.23.

Waite, J., Curzon, P., Marsh, W., and Sentance, S. (2020), 'Difficulties with design: The Challenges of Teaching Design in K–5 Programming', *Computers & Education*, 150, 103838. https://doi.org/10.1016/j.compedu.2020.103838.

20

Misconceptions and the Beginner Programmer

Juha Sorva

Chapter outline

Chapter synopsis

In this chapter, we will review the literature on misconceptions about programming – that is, the intuitive, underdeveloped and possibly flawed ideas that beginners have about specific programming constructs or about the way programs work in general. We will spend much of the chapter looking at examples of common misconceptions and exploring what gives rise to them. We will then briefly view misconceptions through the lens of educational theory before concluding with suggestions on how to address student misconceptions in teaching.

20.1 Introduction

Here are two examples of what we will call *misconceptions*.

M1 A variable can store multiple values; it may store the 'history' of values assigned to it.
M2 Two objects with the same value for a name or id attribute are the same object.

Misconceptions about programming constructs are both common and natural. Even a student that exhibits a misconception usually has some useful knowledge about the concept, but the student is missing a piece, links pieces unproductively or applies their knowledge in an unsuitable context. Through experience and instruction, students overcome these difficulties.

However, since the computer does not negotiate the syntax or semantics of a programming language, misconceptions produce practical problems: a student with a misconception will write programs that do not work. When a misconception persists, it may leave the student frustrated and unable to make progress. Moreover, the student may find it hard to appreciate further instruction and learn from it unless that instruction is sensitive to misconceptions. Research has linked student misconceptions to low self-efficacy in programming (Kallia and Sentance, 2019) and bugs that are difficult to fix (Ettles, Luxton-Reilly and Denny, 2018).

Research in physics education suggests that knowledge of misconceptions is an important component of teachers' pedagogical content knowledge. In a study of hundreds of physics teachers, Sadler et al. (2013) found that those teachers who could identify their students' most common misconceptions were more effective in fostering student learning than those who could not. In computing education, misconceptions research is not yet as well established as in physics education. Nevertheless, there is a substantial and growing body of work that documents misconceptions and demonstrates that some of them occur across individuals and teaching contexts. Much of this work has focused on introductory-level programming, but researchers have also studied misconceptions of data structures and algorithms, program correctness and other computing topics.

Like most studies to date, we will primarily consider text-based programming. Nevertheless, there is evidence that similar misconceptions arise in blocks-based programming as well and that the frequency of particular misconceptions may depend on the chosen format (e.g. Weintrop and Wilensky, 2015; Grover and Basu, 2017).

Many misconceptions arise from a combination of factors that involves the programming content itself, students' prior knowledge and instructional design. Let us begin by reviewing some of these factors. We will see more examples of specific misconceptions as we go.

The examples in this chapter, such as *M1* and *M2* above, have been selected and paraphrased from the hundreds of misconceptions reported in the literature. (See Qian and Lehman, 2017; Sorva, 2012 and references therein.)

20.2 Sources of misconceptions about programming

Mathematics

In many programming languages, dividing the integer 99 by 100 produces zero. This often surprises students but is ultimately just a detail. It is, however, an example of a more general pattern: students bring their mathematics knowledge to the programming class, but the concepts, notations and terms of programming are subtly different from ones that students know from school mathematics.

The variables and assignment statements of (typical imperative) programming look deceptively familiar. Many of the most commonly reported misconceptions are associated with these constructs.

M3 A variable is merely a pairing of a name (symbol) with a value. It is not stored within the computer, apart from the program code.

M4 An assignment statement such as `a = b + 1` stores an equation in memory or stores an unresolved expression `b + 1` in variable `a`.

It is well documented that beginner programmers struggle with sequencing statements; a simple three-line swap of variable values is hard for many students (Lister, 2016). Prior knowledge and notations influence some of these difficulties: even though a sequence of assignment statements is a step-by-step mechanism for manipulating state, it looks much like a set of declarations that hold simultaneously.

M5 A program, especially one with assignment statements, is essentially a group of equations.

M6 Several lines of a (simple non-concurrent) program can be simultaneously active.

Pea (1986) notes that many students find it quite reasonable to sequence a piece of (imperative) code so that the lines that read user inputs follow the line that uses the input data to compute a result. Jimoyiannis (2011) illustrates how some students attempt to 'solve' programs much as one would a group of equations, substituting variable names with the right-hand sides of other assignment statements.

Equations are symmetric, as is the notation `a = b`, but assignment statements are not. It is possible that prior knowledge of mathematics also plays a role in the formation of misconceptions such as the following one.

M7 Assignment statements such as `a = b` work in both directions: they swap the values of two variables.

We have now seen that students sometimes interpret code in terms of mathematics. Whether or not they do so may depend on whether a piece of code looks familiar from math class.

M8 A variable name needs to be a single letter; longer identifiers are interpreted as (parts of) commands.

Grover and Basu (2017) documented an instance of *M8* where students failed to recognize a variable called `NumberOfTimes` as a variable because its name was so long. Instead, the students came up with speculative meanings for the statement where the identifier appeared.

Vocabulary

Some misconceptions arise from the natural-language semantics of words. For instance, some programming languages use the word 'then' exclusively in selection statements: `if a then b else c`. But in English, the word also – and usually – implies a sequence: 'first `a`, then `b`'. Another example is the `while` keyword, which appears in many imperative programming languages. There

is evidence from multiple studies over the past few decades that many students assume the word implies a continuous check:

M9 A `while` loop's condition is evaluated constantly. The instant it becomes `false`, the loop exits.

Students may attribute a similar quality to `if` statements. The English expression 'If you need any help, call me.' does not suggest an immediate, one-time check; it means that the listener ought to call the speaker in the event that they need help later.

M10 An `if` statement triggers whenever its condition becomes `true`.

What is missing from this conception (much as in *M6* above) is the notion that control passes from one instruction to the next. Pea (1986) cites a student who explains: '[The computer] looks at the program all at once because it is so fast.'

Analogies

Teachers employ analogies between programming concepts and more familiar concepts; students also come up with their own analogies and may share them with their peers. Analogies are a useful tool for teaching and reasoning, but they are also a source of misconceptions: since any analogy is a mapping between things that are similar but not identical, there is the risk that learners overextend the analogy.

The classic example from introductory programming is the analogy of a variable being like a box. A variable does indeed have some box-like characteristics, which fact can be used productively in teaching, but the analogy may still produce misconceptions (Hermans et al., 2018), especially if the differences between variables and boxes are not explored. Consider *M1* above, for instance: a box can hold multiple objects at the same time, but a variable only has one. Here are two more examples:

M11 Assignment statements such as `a = b` move values from one variable to another. The source variable is emptied in the process.

M12 Variables are initially empty containers and do not need to be initialized.

Purpose versus structure

Consider the statement `a = a + 1`. Here is how a student, teacher or textbook might describe it: 'It increments the variable `a` by one.' This is a summary of the *purpose* of the line of code, that is, what it accomplishes for the programmer. What was left unsaid is the *structure* of the code: it is an assignment statement, which involves evaluating the composite expression `a + 1` and assigning its value to the variable `a`. That is how the programming environment handles it. Students, however, may simply memorize the 'statement type for incrementing counters' and fail to discern its constituent components. (This is perhaps especially likely if they view assignment statements as equations as in *M4* above, since `a = a + 1` does not fit that interpretation.)

M13 Incrementing a variable is an indivisible operation; no conception of evaluation and assignment.

A piece of code is a causal mechanism that consists of many components. Research on mental models suggests that it is important to be able to reason about a causal system in terms of its individual components without mixing them up with the purpose of the whole (de Kleer and Brown, 1983). It is such an understanding that allows one to debug unexpected behaviour and to combine components in novel ways. Although *M13* is viable for many purposes, it is not felicitous for building a general understanding of program components – expressions and statements.

Programs tend to have a conspicuous line-based format. It may be tempting for students (and teachers) to treat lines as the main constituents of programs and gloss over their internal structure. Often, the concepts of expression and evaluation receive little attention. Here is another example:

```
test = ["A list", "that", "contains", "four strings"]
```

The overall purpose of the line might be summarized as: 'Stores four strings in `test`.' An experienced programmer will not take this too literally. They will perceive that this, too, is an assignment statement, which evaluates the right-hand side and stores the resulting reference in a variable; they will realize that `test` is not really the name of a list object but of a variable. But a beginner may well have more trouble.

M14 Conflation of referring variable with object; the name of the variable is a part of the object. Assignment statements change the names of objects.
M15 An object can be referenced by only one variable.
M16 Declaring a variable also creates an object of the appropriate type.

Let us consider another facet of our example. The description 'Stores four strings in `test`.' is *metonymous*: it refers to an entity, the list of strings, via a structurally associated entity, the name of a variable. Metonymy is pervasive in human communication: we can say things like, 'She likes to read Kafka,' and it will be obvious we refer to the literary works associated with a person. The computer system, however, takes instructions literally and requires us to spell out our purpose in terms of the structural components that the system can manipulate. Beginners may expect the computer to work out what their metonymous instructions mean, which is a potential source of many errors, such as confusing array elements with their indices or attempting to reference objects using the values of their attributes (Miller, 2014).

M17 A field that has distinct values for each object, such as `name` or `id`, works as an identifier for objects.

Metonymy is an example of a still broader source of student difficulties, discussed next.

Expectations of interpretive intelligence

Miller (1981) and Pane, Ratanamahatana and Myers (2001) studied how non-programmers describe procedures in natural language. Among other things, they found that many aspects that are

commonly explicit in programs were implicit or entirely absent in people's natural descriptions: the details of looping, variable declarations, parameter passing, `else` clauses and so on.

In the light of those studies, it is no surprise that beginner programmers are so often taken aback by the level of detail that is required in programming. Beginners frequently assume that what they write says more between the lines than it actually does.

M18 Programs get interpreted more or less like sentences in natural conversation. The computer or programming environment is, for practical purposes, able to deduce the intention of the programmer. It may, for instance, fill in 'obvious' information without being told.

M19 The computer/environment prevents operations that are unreasonable or pointless.

Pea (1986) identified what he termed the *superbug* behind many beginner mistakes: students behave as if 'there is a hidden mind somewhere in the programming language that has intelligent, interpretive powers'. This is not to say that students believe that there is a homunculus running the machine or that computers reason in the same way human brains do. However, when beginner programmers are unsure what to do, they commonly fall back on analogies with familiar forms of language use. Typically, they overestimate the reasoning capabilities of the system. Many scholars have reported on student behaviour consistent with the superbug.

Students sometimes fail to realize that the system does not find meaning in identifiers. For example:

M20 The natural-language semantics of variable names affect which value gets assigned to which variable. For example, `smallest` will surely not store a number greater than the one in `largest`.

Analogies with natural conversation may also influence students' difficulties with specific programming constructs. As one example, `else` clauses are conspicuously rare in non-programmers' process descriptions, as people often neglect alternative branches and may consider them too obvious to merit consideration. This may partially explain why some students have difficulties with `else`, including the following misconceptions.

M21 Using `else` is optional; the code that follows an `if` statement is the `else` branch (if necessary).

M22 Both `then` and `else` branches are always executed.

Intangible concepts beneath program code

Some aspects of programming are more visible than others. In particular, program code – the *static* aspect of programs – is very tangible. It is natural to think about programs in terms of the code that the programmer directly manipulates. However, some of the concepts and processes that explain the runtime behaviour of code – the *dynamic* aspect of programs – are not explicit in code but nevertheless impact what the programmer should do. Examples of the latter group include references, transfer of control between statements, expression evaluation and memory allocation.

Consider functions. The program code that defines and calls them is tangible but function activations at runtime are not. Students commonly find it hard to understand parameter passing, local scope, the lifetime of variables and return values. Some of these difficulties may persist even as students advance in their studies (Fisler, Krishnamurthi and Wilson, 2017). Here are just a few of the reported misconceptions:

M23 Variable names must be different in the calling code and in the function signature. (Alternatively: They must be the same.)
M24 Parameter passing forms direct links between variable names in the call and the signature.
M25 The local variables of methods are members of the object whose method was called. Or vice versa: object members are initialized anew at each method invocation.
M26 Any recursive function is essentially a loop within a single activation of the function.

References, too, are only implicitly present in program code. Students who are unfamiliar with the concept often form misconceptions such as the following. (See also *M14–M17* above.)

M27 Assigning an object to a variable (always) stores the object's properties in the variable.
M28 Assignment statements copy properties from a source object to a target object.
M29 Two objects with identical states are the same object.

Another example is the relationship between a class and its instances. In many programming languages, classes are tangible, static definitions, whereas object creation happens dynamically at runtime. This is a notoriously difficult idea in introductory object-oriented programming, as are the related concepts of the constructor and the `this` (or `self`) reference (see, e.g., Holland, Griffiths and Woodman, 1997; Ragonis and Ben-Ari, 2005).

M30 An object is essentially just a piece of code, difficult to distinguish from a class.
M31 A class is a collection of – or container for – objects.
M32 Instantiation involves only the execution of the constructor body, not the allocation of memory.
M33 Providing a constructor definition is sufficient for object creation to happen.
M34 `this` is the class in which a method is implemented.

An implicit execution model

Many or even most of the misconceptions that we have discussed so far illustrate a common point: students often struggle to predict the behaviour that results when the computer executes the instructions in a program. Many students lack a way of reasoning that enables them to reliably answer questions such as: What can this programming system do for me? What are the things it can't or won't do? Which rules does it follow? What is the division of labour between myself and the computer system – that is, between the human instructor and the mechanistic instructee? What changes within the system does each of my instructions bring about?

In other words, students need to be taught a *model of program behaviour* (Duran, Sorva and Seppälä, 2021), an abstraction of computer software and hardware that suits the sort of programming that students will do.

Different programming languages and environments call for different models of program behaviour. For instance, a model that explains the behaviour of Python programs may involve concepts such as memory allocation, flow of control, references and call stack. A model for pure functional programming may be simpler.

Even a single programming language can be viewed in terms of different models. One model may be more or less detailed than another, or more consistent, technically accurate, generalizable to other languages and so forth (Duran, Sorva and Seppälä, 2021). Which model works best will depend on learning goals and context.

Models of program behaviour are not always explicitly discussed by textbooks, teachers or students. This makes it likelier that students fail to construct reliable knowledge.

Limited variation in programs

Students commonly cite concrete experiences with programs as their main source of programming knowledge. Whether from given examples or programs they write themselves, students infer implicit 'rules' from the programs they encounter. Limited exposure to different programs can lead to under- or overgeneralized rules that limit students' ability to make use of programming constructs. Here are some examples of misconceptions that unnecessarily restrict the programmer:

M35 Object attributes must be numbers or similarly primitive data.
M36 Function arguments must be literals or constants.
M37 Comparisons must appear within a conditional expression and not, for example, in `return` or assignment statements. Booleans are perceived as parts of control structures, not as values in the same sense as numbers.
M38 Only one instance can be created for each class.
M39 A method can be invoked only once (on each object, or in total).
M40 A class can have only one method./A class can have only one member variable.

Misconceptions such as *M8*, *M17* and *M20*, from earlier in this chapter, indicate that some students find it difficult to understand how the programming environment handles identifiers. Indeed, students may find it difficult to tell the difference between an identifier chosen by the programmer and a construct of the programming language; some learners search the internet for variable names and other program-specific identifiers as they attempt to find help for programming problems. Limited variation in programs can exacerbate these difficulties. For instance, if all or most objects have a `name` or `id` attribute, students are more likely to think of the identifier as part of the language, as in *M17*. Another example, from personal experience: in a course where I frequently used `another` as the parameter name in methods that compare the active object with the given object, some students ended up thinking `another` was a reserved word akin to `this`.

Teachers

As noted at the beginning of this chapter, teachers' knowledge of students' likely misconceptions makes a difference to student learning. In addition, the same study by Sadler et al. (2013) shows that teachers' own subject-matter knowledge is a necessary (but insufficient) precondition of knowledge of misconceptions. In other words, teachers who 'know their stuff' are less likely to encourage misconceptions in their students, especially if the teacher additionally knows how 'the stuff' challenges learners.

Other aspects of teachers' ability and instructional design also matter, of course. For instance, excessively complex or unmotivating tasks will result in poorer learning and may contribute to misconceptions.

20.3 Some theoretical perspectives on misconceptions

Misconceptions have long been of interest to researchers of *conceptual change* (Vosniadou, 2008). Theories of conceptual change seek to characterize the nature of people's conceptual structures and how those structures change as we learn.

Classical conceptual change theory posits that conceptual structures are 'theory-like' in that even intuitive knowledge is relatively organized and structurally coherent. Moreover, people are generally averse to disturbing this relatively harmonious state, which explains the resilience of existing conceptions. In the classical view, misconceptions are often seen as obstacles that need to be avoided or confronted; learning results from cognitive conflict between an existing conception and the demands of a novel situation, leading to the rejection of the existing conception and its replacement with a new one. By designing situations that engender cognitive conflict, teachers can help students overcome their flawed conceptions.

Knowledge-in-pieces theories of conceptual change reject the classical view. Their proponents argue that intuitive knowledge has no coherent overall organization and consists of largely isolated, highly context-dependent elements. It is both possible and common for a person to have multiple apparently contradictory ideas about a phenomenon so that they draw on whichever idea occurs to them in a particular situation. A programming student, for example, might draw on distinct conceptual structures for:

- Assignment statements with objects vs assignment statements with primitive values (Sorva, 2008),
- Scalar variables vs array variables (Lister, 2016),
- Variables that store numbers vs ones that store strings (Hermans et al., 2018) or
- Assignment statements that increment a variable (*M13*) vs other assignment statements (*M4* & *M5*).

It is the fragmented nature of knowledge that makes misconceptions resilient: a learner can add a parallel understanding rather than supplanting an existing one.

Knowledge-in-pieces theories encourage an emphasis on the productive parts of learners' existing conceptions. Learning is seen as being not about confronting flawed conceptions but about connecting and organizing intuitive ideas and discovering to what extent they are viable and how they can be developed. Elements of intuitive knowledge – so-called misconceptions included – are raw materials for learning that evolve into more general, theory-like structures.

The neo-Piagetian perspective of Lister (2016) complements knowledge-in-pieces theories of conceptual change. Lister describes how it is normal for beginner programmers to initially exhibit a low level of commitment to their conceptions about programming, to routinely swap between conceptions and to develop error-prone *ad hoc* tracing strategies for different programs.

20.4 Implications for pedagogy

A model of program behaviour as a learning objective

One of the foremost recommendations from the misconceptions literature is that teachers need to help students reason about program behaviour. This does not imply that a low-level hardware model is necessary; students can be taught a more abstract model of program behaviour that operates just beneath the level of program code that the students manipulate. The key is to identify what the students need to know so that they can program and learn successfully.

This means that students may need to learn concepts that are not directly visible in program code but that help explain the behaviour of programs, such as expression evaluation, memory allocation, references or the call stack. These concepts can be introduced alongside new programming constructs, gradually extending a simple initial model. Duran, Sorva and Seppälä (2021) present an example progression of 'rules of program behavior' for Python programs, discuss the related trade-offs in instructional design and elaborate on the need to teach a model of program behaviour. Fincher et al. (2020) illustrate a collection of pedagogic devices ('notional machines') that various teachers have used to explain how programs behave.

This is not a call to add more content to already crowded curricula. It is a call to acknowledge content that is already in but often neglected. Any programming environment has some underlying behaviour that students will need to cope with. Without help, students are forced to rely on guesswork, fragile analogies and mistaken assumptions, which may prohibit productive programming and effective learning.

Advice for misconception-sensitive teaching

Try to learn to see programming concepts and language constructs from your students' perspectives. Discussions, observation and collecting feedback can all help in this, but you may need to probe deep: 'It was not until we did the tedious work of having students walk through every command in a program, thinking aloud and explaining how the computer would interpret it, that we became aware of the prevalence of these [conceptual] bugs. After that, we saw them everywhere' (Pea, 1986).

Expect students to conceptualize code constructs in unexpected ways. Expect conceptions that are fragmented and may appear inconsistent with each other. Reflect on your own assumptions; do not assume that students share them. Draw on the research literature to prepare for common misconceptions that your students may develop. Where possible, use concept inventories or similar assessments (Taylor et al., 2014; Parker and Guzdial, 2016; Grover, 2020) to assess students' prior knowledge and the effectiveness of your teaching.

Find the valuable aspects of students' conceptions and build on them. When you encounter a misconception, remember to consider the situations where the conception *is* viable. Guide students to explore the viability of their knowledge for different kinds of programs and to compare and contrast different conceptions. Obviously, do not dismiss misconceptions as failures; instead, help your students emotionally so that they are not discouraged by the misconceptions.

One does not learn to program without program-writing practice, but don't take that to mean that the most effective way to learn is *only* to write programs; to learn to write code, students also need to learn to read code (Lister, 2016; Xie et al., 2019). Don't assume that because a student produced a piece of code, they understand it (Salac and Franklin, 2020). Design code-reading activities that target a model of program behaviour. Use worked-out examples and case studies of programs (Skudder and Luxton-Reilly, 2014; Linn and Clancy, 1992). Have students fix or extend given programs, which motivates a careful study of the given code and lets students work on more interesting applications than they could write from scratch.

Encourage students to explain program behaviour in detail to themselves and to others. Arrange for students to get feedback on their explanations. Consider peer instruction (Porter, Bailey-Lee and Simon, 2013) and other collaborative activities as a way to explore and address misconceptions. Design classroom discussions around apparent or expected student misconceptions (Ginat and Shmallo, 2013).

Teach students self-explanation skills (Fonseca and Chi, 2011) so that they can focus on underlying principles rather than the surface features of a particular program. Be aware that learning to trace programs requires repeated practice; have students practice tracing and present general principles as feedback on their tracing attempts (Lister, 2016).

Teach students to explain both the overall purpose of a piece of code (the forest) and its individual constituents (the trees). Watch out for explanations that mix purpose with structure, both in what you say and in what students say. Teach a model of program behaviour that has sufficient granularity (Duran, Sorva and Seppälä, 2021): highlight that lines of code are not atomic; acknowledge expressions and evaluation as significant concepts.

In popular constructionist pedagogies, students use tools such as Scratch to build and share fun creations but may fail to learn foundational concepts; complement constructionism with other approaches that target conceptual understanding (Grover and Basu, 2017).

The example programs that students encounter are a major source of their programming knowledge; strive for exemplary examples. Use lists of known misconceptions as inspiration for the design of examples and related activities. Try to make sure that students encounter rich variation in programs to prevent particular misconceptions: Did my example class have just one instance or just one method (*M38*, *M40*)? How have I highlighted to my students that the execution order of statements makes a difference (*M5*)? Have I used multiple variables referencing a single object

and a single variable referencing a succession of objects (*M14*, *M15*)? Have the students had to deal with multiple identical but distinct objects (*M29*)? Do my examples of `while` discourage the 'continuous evaluation of condition' interpretation (*M9*)? Do I have an example that shows how changing the value of a variable does not impact the values of other variables, even those initialized using it (*M4*)? Have I included an example of multiple variables with the same name in different scopes? What about nested function calls and other complex function arguments (*M36*)? And so on.

Pay particular attention to examples that introduce a concept. Note that using the simplest imaginable example may give a misleading first impression (e.g. the first variable students see is assigned an integer literal). Do not use only code that works; make use of erroneous programs as well. Use combinations of examples with minimal differences to separate aspects and highlight contrasts.

Have students view, draw or manipulate visualizations of hidden processes, or role-play the processes. Consider program visualization software designed for education (Sorva, Karavirta and Malmi, 2013). Recognize that reading a visualization is itself a skill that needs to be learned and that struggling beginner programmers may struggle to decipher the visualizations, too.

Analogies and metaphors can provide viable alternatives to technical terms; look for consistent analogies that explain many concepts and their relationships (e.g. Gries, 2008). Reflect on the borders of any analogies you use and deliberately explore those borders with your students. Watch out for known issues with common analogies (e.g. variable as box).

Teach students to spell things out literally enough when they instruct the computer. Discuss the differences between programming and human conversation – even human conversation that is about programming. Explore natural-language phenomena such as metonymy in an age-appropriate manner. Consider what other prior knowledge, besides language, might be leveraged or otherwise taken into account (see, e.g., Simon et al., 2008).

Be on the lookout for deceptively familiar notations and terms from mathematics and their subtle impact on student understandings. Explicitly point out some of the most common pitfalls.

Don't forget about program-writing practice either.

Key points

- Beginner programmers commonly struggle with specific programming constructs and the nature of programs more generally.
- Factors that contribute to student misconceptions include: limited exposure to different programs, intangible concepts and an implicit model of program behaviour, students' prior knowledge of mathematics and natural language and the analogies employed in teaching.
- To teach most effectively, teachers should be familiar both with the programming concepts and with how students commonly view the concepts.
- Teachers should make it their goal to teach students a model of program behaviour, a reliable basis for reasoning about what program code does when executed.

- Teachers can help students construct increasingly productive knowledge by providing ample practice with reading and tracing a rich variety of programs, designing examples to target common misconceptions, bringing the intangible aspects of programming into focus and probing the borders of analogies and metaphors.

For further reflection

- Review a programming textbook or the materials you use for teaching. Consider whether the text, examples and activities are likely to help with – or encourage – the misconceptions listed in this chapter.
- Reflect on the model of program behaviour that your students need. What capabilities of the programming environment are invoked by the commands in the language? What do the students need to know about those capabilities in order to reason reliably about program behaviour? Which 'rules' does the system obey?

References

De Kleer, J., and Brown, J. S. (1983), 'Assumptions and Ambiguities in Mechanistic Mental Models', in D. Gentner and A. L. Stevens (eds), *Mental Models*, 155–90, Hillsdale, NJ: Lawrence Erlbaum.

Duran, R., Sorva, J., and Seppälä, O. (2021), 'Rules of Program Behavior', *ACM Transactions of Computing Education*, 21 (4): 1–37.

Ettles, A., Luxton-Reilly, A., and Denny, P. (2018), 'Common Logic Errors Made by Novice Programmers', in *Proceedings of the 20th Australasian Computing Education Conference (ACE'18)*, 83–9, ACM.

Fincher, S., Jeuring, J., Miller, C. S., Donaldson, P., Du Boulay, B., Hauswirth, M., Hellas, A., Hermans, F., Lewis, C., Mühling, A., Pearce, J. L., Petersen, A. (2020), 'Notional Machines in Computing Education: The Education of Attention', in Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education, 21–50.

Fisler, K., Krishnamurthi, S., and Wilson, P. T. (2017), 'Assessing and Teaching Scope, Mutation, and Aliasing in Upper-Level Undergraduates', in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, *SIGCSE '17*, 213–8, ACM.

Fonseca, B. A., and Chi, M. T. H. (2011), 'Instruction Based on Self-Explanation', in R E Mayer and P A Alexander (eds), *Handbook of Research on Learning and Instruction*, 296–321, New York: Routledge.

Ginat, D., and Shmallo, R. (2013), 'Constructive Use of Errors in Teaching CS1', in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE '13*, 353–8, ACM.

Gries, D. (2008), 'A Principled Approach to Teaching OO First', *SIGCSE Bulletin*, 40 (1): 31–5.

Grover, S. (2020), 'Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic', in *Proceedings of the 2020 ACM SIGCSE Technical Symposium on Computer Science Education, SIGCSE '21*, 678–84, ACM.

Grover, S., and Basu, S. (2017), 'Measuring Student Learning in Introductory Block-Based Programming: Examining Misconceptions of Loops, Variables, and Boolean Logic', in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, SIGCSE '17*, 267–72, ACM.

Hermans, F., Swidan, A., Aivaloglou, E., and Smit, M. (2018), 'Thinking Out of the Box: Comparing Metaphors for Variables in Programming Education', in *Proceedings of the 13th Workshop in Primary and Secondary Computing Education, WiPSCE '18*, 1–8, ACM.

Holland, S., Griffiths, R., and Woodman, M. (1997), 'Avoiding Object Misconceptions', *SIGCSE Bulletin*, 29 (1): 131–4.

Jimoyiannis, A. (2011), 'Using SOLO Taxonomy to Explore Students' Mental Models of the Programming Variable and the Assignment Statement', *Themes in Science and Technology Education*, 4 (2): 53–74.

Kallia, M., and Sentance, S. (2019), 'Learning to Use Functions: The Relationship between Misconceptions and Self-Efficacy', in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE '19*, 752–8, ACM.

Linn, M. C., and Clancy, M. J. (1992), 'The Case for Case Studies of Programming Problems', *Communications of the ACM*, 35 (3): 121–32.

Lister, R. (2016), 'Toward a Developmental Epistemology of Computer Programming', in *Proceedings of the 11th Workshop in Primary and Secondary Computing Education, WiPSCE '16*, 5–16, ACM.

Miller, C. S. (2014), 'Metonymy and Reference-Point Errors in Novice Programming', *Computer Science Education*, 24 (2–3): 123–52.

Miller, L. A. (1981), 'Natural Language Programming: Styles, Strategies, and Contrasts', *IBM Systems Journal*, 20 (2): 184–215.

Pane, J. F., Ratanamahatana, C. A., and Myers, B. A. (2001), 'Studying the Language and Structure in Non-programmers' Solutions to Programming Problems', *International Journal of Human–Computer Studies*, 54 (2): 237–64.

Parker, M. C., and Guzdial, M. (2016), 'Replication, Validation, and Use of a Language Independent CS1 Knowledge Assessment', in *Proceedings of the 12th International Computing Education Research Conference, ICER '16*, 93–101, ACM.

Pea, R. D. (1986), 'Language-Independent Conceptual "Bugs" in Novice Programming', *Journal of Educational Computing Research*, 2 (1): 25–36.

Porter, L., Bailey-Lee, C., and Simon, B. (2013), 'Halving Fail Rates Using Peer Instruction: A Study of Four Computer Science Courses', in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education, SIGCSE '13*, 177–82, ACM.

Qian, Y., and Lehman, J. (2017), 'Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review', *ACM Transactions on Computing Education*, 18 (1): 1–24.

Ragonis, N., and Ben-Ari, M. (2005), 'A Long-Term Investigation of the Comprehension of OOP Concepts by Novices', *Computer Science Education*, 15 (3): 203–21.

Sadler, P. M., Sonnert, G., Coyle, H. P., Cook-Smith, N., and Miller, J. L. (2013), 'The Influence of Teachers' Knowledge on Student Learning in Middle School Physical Science Classrooms', *American Educational Research Journal*, 50 (5): 1020–49.

Salac, J., and Franklin, D. (2020), 'If They Build It, Will They Understand It? Exploring the Relationship between Student Code and Performance', In *Proceedings of the ACM Conference on Innovation and Technology in Computer Science Education,* ITICSE '20, 473–9, ACM.

Simon, B., Bouvier, D. J., Chen, T.-Y., Lewandowski, G., McCartney, R., and Sanders, K. (2008). 'Common Sense Computing (Episode 4): Debugging'. *Computer Science Education*, 18 (2): 117–33.

Skudder, B., and Luxton-Reilly, A. (2014), 'Worked Examples in Computer Science', in J. Whalley and D. D'Souza (eds), in *Proceedings of the 16th Australasian Conference on Computing Education (ACE '14),* vol. 148 of CRPIT, 59–64, Australian Computer Society. ACM.

Sorva, J. (2008), 'The Same but Different—Students' Understandings of Primitive and Object Variables', in A. Pears and L. Malmi (eds), *The 8th Koli Calling International Conference on Computing Education Research*, Koli Calling '08, 5–15, Uppsala University.

Sorva, J. (2012), 'Misconception Catalogue', *Visual Program Simulation in Introductory Programming Education*, 358–68, Doctoral Dissertation, Department of Computer Science and Engineering, Aalto University.

Sorva, J., Karavirta, V., and Malmi, L. (2013), 'A Review of Generic Program Visualization Systems for Introductory Programming Education', *ACM Transactions on Computing Education*, 13 (4): 1–64.

Taylor, C., Zingaro, D., Porter, L., Webb, K., Lee, C., and Clancy, M. (2014), 'Computer Science Concept Inventories: Past and Future', *Computer Science Education*, 24 (4): 253–76.

Vosniadou, S. (ed.) (2008), *International Handbook of Research on Conceptual Change*, New York: Routledge.

Weintrop, D., and Wilensky, U. (2015), 'Using Commutative Assessments to Compare Conceptual Understanding in Blocks-Based and Text-Based Programs', in *Proceedings of the Eleventh Annual Conference on International Computing Education Research, ICER '15*, 101–10, ACM.

Xie, B., Loksa, D., Nelson, G. L., Davidson, M. J., Dong, D., Kwik, H., Tan, A. H., Hwa, L., Li, M., and Ko, A. J. (2019), 'A Theory of Instruction for Introductory Programming Skills', *Computer Science Education*, 29 (2–3): 1–49.

21

Programming in the Classroom

Sue Sentance and Jane Waite

Chapter outline

Chapter synopsis

In this chapter, we follow on from Chapter 18, 'Principles of Programming Education', to describe practical, research-backed approaches that can be used in the programming classroom. We describe some of the difficulties faced by young programmers and then highlight research relating to three broad areas of collaboration, code comprehension and modelling. Moving to the specific application of pedagogies, we next outline specific and popular contexts of physical computing, block-based programming, project-based learning, unplugged programming and games. We finish by considering specific ways to support students to learn to program, including scaffolding the learning and incorporating a range of strategies to support a deeper understanding of programming.

21.1 Introduction

Programming is a key part of computer science and computing; it is a skill that cannot sit separately from the theoretical components of computing. Rather, programming is the application of concepts that are often hard to understand until they are put into practice. Programming practice is not simply skill reinforcement; it is the route to understanding.

To teach any subject requires good teaching skills, knowledge about the subject being taught and specific knowledge – known as pedagogical content knowledge – that a teacher gains about how to teach a particular topic to their students in the learning context at a given moment in time. When reading this chapter, you might wish to think carefully about which combination of instructional approaches is likely to ensure that learning is accessible for all your students.

21.2 Why do learners struggle with programming?

Learners new to programming can find it difficult. For example, it has been asserted that beyond the syntax and semantics of particular programming concepts, beginner programmers may struggle to put these together to construct a program (Robins, Rountree and Rountree, 2003). Actually writing code (as opposed to reading) is particularly hard for novice programmers (Qian and Lehman, 2017), and it is commonly believed that code tracing is easier than code writing (Denny, Luxton-Reilly and Simon, 2008). However, many students find code tracing challenging (Vainio and Sajaniemi, 2007) with particular difficulties being around single value tracing, confusion of function and structure, external representations and levels of abstraction.

The mental effort needed by learners as they embark on this complex journey of learning to program can also be viewed through cognitive load theory (van Merriënboer and Sweller, 2005). Cognitive load theory is a theory of instructional design that suggests that some instructional techniques assume a processing capacity greater than our limits and so are likely to be defective and that students should instead engage in activities that are directed at schema acquisition and automation (Sweller, 1994). Working independently on programming has been suggested to have higher cognitive load than working collaboratively through pair programming (Tsai, Yang and Chang, 2015).

However, we may inadvertently use teaching methods that don't help this situation at all. A reliance on programming textbooks and 'show me' approaches to teaching coding means that novices may end up being asked to copy in a section of code that has no meaning to them at all. Add this to the fact that younger learners will be developing their literacy and keyboard skills, the process of copying in can be incredibly frustrating and dispiriting. Another practice might be to model writing a program from the front while learners watch and then ask learners to go ahead and write a similar program themselves: this also leaves a huge chasm for the novice programmer to fill in themselves, which many simply cannot manage.

21.3 Strategies for teaching programming

Any teacher, of any subject, will draw on a toolkit of teaching strategies to support students in their learning and provide variety in lessons. Teaching programming is no different, and we encourage teachers to develop their own toolkit of strategies and decide what is appropriate for each teaching context. Here, we outline some of the research underlying some common approaches to the teaching of programming.

Code comprehension

For many years, researchers have highlighted the importance of reading code and being able to trace what it does before writing new code (Lister et al, 2004). Comparing tracing skills to code writing, they demonstrated that novices require a 50 per cent tracing code accuracy before they can independently write code with confidence (Venables, Tan and Lister, 2009). Learning to program is sequential and cumulative, and tracing requires students to draw on accumulated knowledge to conceive a big picture. Work by Teague and Lister in this area suggests that novice learners should be focused on very small tasks with single elements (Teague and Lister, 2014). Another study concluded that as well as inferring meaning from code from its structure, the first step should be to make inferences about the execution of the program (Busjahn et al., 2013).

There is a wide range of potential code comprehension tasks that can be used as learning activities to highlight students' alternate conceptions or exemplify programming concepts. Some examples are spotting concepts, recalling facts or examples, changing aspects of programs and comparing and decomposing solutions. More specifically, students can predict what code will do, match designs to programs, investigate and fix buggy code or sabotage code for their peers to fix. Students can be asked to annotate code with an explanation of what the code is intended to do. Parts of code can be removed and students asked to fill the gaps.

One particular activity to support code comprehension is Parson's problems; these provide learners with all the code required, but in sections and with the sections in the wrong order (Parsons and Haden, 2006). There are many variants of Parson's problems, such as including superfluous lines of code with common syntactic or semantic errors to act as distractors, faded Parson's problems where students increasingly complete some lines of code and adaptive Parson's problems, which dynamically control problem difficulty based on a student's performance. Parson's problems have been suggested to be particularly effective in helping university students understand patterns in programs (Weinman et al., 2021) and improving student engagement (Ericson, Margulieux and Rick, 2017).

Languages like Python (commonly used in schools in England) are often celebrated because you can write a program in a short number of lines. However, this can often mean there are lots of programming concepts to understand in one line of code. One way to unpack what the code is doing is to align comprehension exercises to the Block Model (Schulte, 2008; Izu et al., 2019). The Block Model, as shown in Table 21.1, distinguishes between a new programmer's understanding

Table 21.1 The Block Model

(**M)** Macro structure	Understanding the overall structure of the program text	Understanding the algorithm underlying a program	Understanding the goal/purpose of the program in the current context
(R) Relationships	Relationships between blocks	Sequence of function calls, object sequence diagrams	Understanding how subgoals are related to goals
(B) Blocks	Regions of interest that build a unit (syntactically or semantically)	Operation of a block or function	Understanding of the function of a block of code
(A) Atoms	Language elements	Operation of a statement	Function of a statement
	(T) Text surface	**(P)** Program execution	**(F)** Function
	Architecture/structure		Relevance/intention

Source: Adapted from Schulte (2008).

of the structural atomic detail of a program, the code, the functional goals of the program and the problem (Schulte, 2008).

Izu et al. (2019) mapped a set of code comprehension activities to the Block Model. Some of these are shown in Table 21.2, adapted a little from the original paper.

Example: Investigate questions

Use some of the activities from Table 21.2 to encourage students to investigate an extract of program code. For example:

- Ask the question, 'What would happen if those two lines were the other way round?'
- Ask the question, 'What would happen if the input to the program was ____?'
- Ask students to draw on the program to identify blocks of code or the type of a construct?
- Ask students to draw the flow of control on the program showing what line is executed when a loop is exited.
- Ask students to identify the scope of a variable.

Table 21.2 Example code comprehension activities aligned to the twelve-cell Block Model

	Text Surface	**Program Execution**	**Function**
Macro	• Describe the overall program block structure by drawing nested boxes • Represent the overall program structure by drawing a tree of function/procedure dependencies	• Verify if a program statement or block is ever reachable during program execution • Identify a comprehensive set of inputs to check all possible computation flows of a program	• Choose an appropriate name for a program • Create meaningful test cases for the allowed inputs and expected outputs
Relational	• Link each occurrence of a variable with its declaration • Identify where a particular function is called	• Trace the program execution for a given input, where the program includes calls to procedural units • Verify whether some branches of a switch/case statement are redundant, that is, can never be executed • Identify the scope of a variable	• Choose an appropriate name for a variable • Solve a Parson's puzzle for a given code purpose by reordering simple blocks
Block	• Draw a box around the code of each conditional construct • Draw a box round the body of each method/ procedure/function	• Identify recurring instrumental blocks such as that for swapping the values of two variables • Identify the block(s) implementing some specific program pattern	• Summarize in a short sentence what the block goal is • Identify the program block(s) with a given function, described in problem-domain terms
Atom	• List all integer variables • Draw a box around the headers of all procedures/functions in a piece of code	• Determine the program output for given input data, again where the program does not include procedural units • Determine the value of an expression for given values of the involved variables	• Identify the purpose of an expression or a simple statement, in connection with the problem domain • Rename a constant with an appropriate name from the problem

Source: Adapted from Izu et al. (2019).

Collaboration

As we have seen in Chapter 14, dialogue and classroom talk are an important aspect of the teaching and learning of programming. When learning to program, students can be encouraged to discuss their code with one another. Collaborative, program-focused tasks involving talking about programs can help the development of socially constructed knowledge about programming. There

are a number of ways in which we can encourage collaboration when teaching programming. Two of the most popular approaches are *pair programming* and *peer instruction*.

Pair programming is a well-researched method for engaging students in programming, with evidence that it can improve teaching and learning. However, care needs to be taken when implementing this approach with social dynamics, power struggles, friendship dynamics, confidence with computers, inequity of roles, how students talk to each other, the structure of tasks and teacher intervention, all potentially impacting interactions and learning (Shah and Lewis, 2019; Denner, Green and Campe, 2021). More research with larger numbers of students in different contexts with carefully controlled interventions is needed to provide robust recommendations for classroom practice.

Key concept: Pair programming

Used in industry and education, pair programming is a collaborative approach where two people simultaneously work on a single software development project. Swapping roles regularly, one person (the driver) has control of the mouse and keyboard, and the other (the navigator) continuously collaborates by reviewing the written code and keeping track of work done against the design (McDowell et al., 2006).

Key concept: Peer instruction

Peer instruction (PI) is not simply peers teaching each other. In class, learners are provided with carefully constructed, concept-based, multiple-choice questions, which are based on pre-lesson reading. Learners independently consider the questions and give their answer (vote) using flashcards or an online voting system. They then share their responses with their peers and discuss their thinking before re-submitting their answer (re-vote). The teacher reviews learners' first and second answers and, if needed, provides further support after the second answers before moving on to the next question.

Example: Activities to encourage collaboration

- Develop a pair working agreement for your own class and use it when introducing pair programming. Experiment with different ways of pairing students and reflect on what is most successful for your own teaching context.
- Try peer instruction, including asking your students to reflect on the experience and whether it supported their learning.

- Encourage students to talk through their program with another student, paying attention to the terms that they use and what they find difficult to explain. Ask students whether talking through a program with another person helps them with troubleshooting.

Modelling

Modelling is a form of in-class demonstration where students observe as a teacher completes an activity whilst talking through their thought process. This brings an apprenticeship approach to teaching, and in the teaching of programming, it is also referred to as live coding (Rubin, 2013) (not to be confused with 'live coding' as a form of performance art). Pupil interaction may be introduced into this approach by asking learners what to do next at various points in the activity and by asking them to spot mistakes. Modelling is also used in the teaching of other subjects and can be a very useful approach to introduce programming concepts and processes.

To support learning through live coding, two things are essential. First, the teacher must carefully select appropriate examples for teaching new concepts, consolidating understanding or addressing existing or potential misconceptions. Second, live coding should reveal the thinking of the demonstrator: what the teacher says as they 'think aloud' is crucial to the effectiveness of live coding.

As well as using sample programs for students to predict what the program will do, teachers also use sample programs when they model how to write code based on worked examples. Worked examples can be provided to students for them to learn about concepts, processes and features of programming environments, such as the concept of iteration, the process of development, the role of variables and tools for debugging. A further enhancement of the use of worked examples is subgoal modelling, whereby meaningful labels are added to worked examples to visually group steps into subgoals, highlighting the structure of code. Recently, Margulieux et al. (2020) redesigned a resource aimed at 15- to 18-year-olds, adding subgoal labels to a set of resources used in an Advanced Placement programming course and found some positive effects on outcomes. Students learning with subgoals performed no better in knowledge-based assessment but performed better on problem-solving questions, wrote more on open-ended questions and continued to use subgoals after the course. Teachers in the study suggested that struggling students found subgoals the most useful (Margulieux et al., 2020).

Putting it all together

One approach to teaching programming which emphasizes code comprehension, student collaboration and teachers' use of language when modelling is PRIMM (Sentance, Waite and Kallia, 2019). PRIMM is an approach that can help teachers structure lessons in programming. PRIMM stands for predict, run, investigate, modify and make, representing different stages of a lesson or series of lessons. PRIMM promotes discussion between learners about how programs work and the use of starter programs to encourage them to read code before they write it. A number of studies have been employed to investigate the impact of PRIMM; the largest of these was a mixed

methods study involving around five hundred students aged eleven to fourteen. The results showed that PRIMM lessons had an impact on programming attainment and that teachers particularly value the collaborative approach, the structure given to lessons and the way that resources can be differentiated (Sentance, Waite and Kallia, 2019).

Example: Teaching the PRIMM way

Try PRIMM out in a programming lesson. You may need a sequence of lessons to complete all five steps, with some iteration of predict, run and investigate before students are ready for modify and make.

- *Predict:* Give students a program (on paper or on the board) and ask them to discuss it and predict what it might do. Students can draw or write out what they think will be the output.
- *Run:* Give students an executable version of the program. Ask students to run the program so that they can test their prediction. Discuss the results with the class.
- *Investigate:* Provide a range of activities to explore the structure of the code, such as tracing, explaining, annotating, debugging and so on. These can be mapped to the Block Model as in Table 21.1.
- *Modify:* Provide a sequence of increasingly more challenging exercises to enable students to edit the program to change its functionality; the transfer of ownership moves from the code being 'not mine' to 'partly mine' as students gain confidence by extending the function of the code.
- *Make:* Set a new programming problem for the students. Support the students to plan and design a solution to the problem. Developing the solution to the new problem should enable students to practice the programming skills they learned during the previous steps.

21.4 Contexts and environments

In this section, we move from strategies in the teachers' toolkit to the variety of contexts and environments that can be used to learn programming.

Block-based programming

Sometimes called block-based, visual or graphical programming languages, these languages use graphical images to represent programming commands. These easy-to-use languages are used not only with the youngest learners in formal and non-formal learning contexts but also with older students in formal introductory programming lessons.

Block-based languages and their programming environments provide a range of affordances over and above text-based languages. Affordances include not requiring students to memorize and type in commands or deal with unfamiliar and sometimes confusing characters such as {}, [] and == and presenting natural language-type block labels. Commands are often grouped by colour to give hints about their shared purpose, and shapes dynamically change their size to signal the scope of the command. Common shapes indicate which combinations of programming objects are allowed and provide an environment that allows quick and easy program-building (Bau et al., 2017; Weintrop et al., 2019).

Unplugged programming

Unplugged activities teach about computing without a computer (Bell et al., 2009). Role-play can provide a physical enactment of a complex concept. For example, acting out a bubble-sort breaks down the process into individual steps and highlights features that might otherwise be difficult to envisage (Katai, Toth and Adorjani, 2014). Role-play can also be used to help learners design new products as they step through and try out their ideas. For example, when learning how to program programmable toys, students can 'play turtle' to help them understand the way the machine works as they embody and execute the steps of their solution (Papert, 1980).

Examples of unplugged activities and how to use the semantic wave theory to introduce them effectively are given in Chapter 13 on concepts.

Physical computing

Physical computing (also called tangible computing) refers to the use of both software and hardware to build interactive physical systems that sense and respond to the real world. It includes building tangible interactive objects or systems, designing with creativity and imagination and engaging physically as well as mentally. From a learning perspective, physical computing intersects a range of activities often associated with design technology, electronics, robotics and computer science. But perhaps more importantly, physical computing provides a means to explore the use of technology in a wide range of subjects.

Physical computing can be very useful in the learning of programming. With a physical device, there can be instant feedback from the device to indicate whether the program code works as desired. This can be rewarding as well as accelerate learning.

A common finding from research is that physical computing projects are particularly motivational to pupils (Garneli, Giannakos and Chorianopoulos, 2015). There are many different types of physical computing devices. Devices can include packaged electronics with no programming required, programmable robots and construction sets, programmable boards with integrated or external input and output devices that need a PC during use, battery-powered embedded programmable boards which can operate without a PC and general-purpose programmable boards that often use wired power (Hodges et al., 2020).

Game creation

Game creation is often used as a context for learning how to program. However, as with research into physical computing pedagogy, the evidence for game creation being beneficial is often not robust (Kafai and Burke, 2015). One notable example of a games context being used to teach programming is the work of the Scalable Design Team (Repenning et al., 2015). Rather than basing teaching on objectives related to students learning about programming constructs such as sequence, selection and repetition, the curricula focus on common patterns used in creating simulations.

Across all contexts, *project-based learning* can be used to give learners autonomy, ownership and realism as learners are provided with choices as to what to investigate and build (Thomas, 2000). A project-based learning approach to programming is associated with construction and constructionism as learners make things (or knowledge) through active exploration (Papert, 1980) and where the products made are meaningful in some way to the maker (Kafai and Resnick, 1996). This approach has come under criticism (e.g. Mayer, 2004), and incorporating scaffolding can be useful to provide some structure (Lye and Koh, 2014).

21.5 Supporting learners

Programming involves a complex set of skills, and some learners may find it challenging. There is a range of ways to support learners as they gradually master the key concepts and practice the skills.

Cognitive apprenticeship

Cognitive apprenticeship is a concept introduced back in the 1980s and refers to how novices can gain expert skills by observing and then practising expert activity (Collins, Brown and Newman, 1987). Some teaching approaches associated with cognitive apprenticeship are modelling, coaching, scaffolding, student articulation, reflection and exploration. In their review of teaching and learning of computational thinking through programming, Lye and Koh found that authentic contexts with scaffolding and reflection activities appeared to be the most successful, but the authors advised that no one pedagogical solution is appropriate for all classes. They suggested using a number of approaches that fall under the umbrella of cognitive apprenticeship, including much scaffolding at the start of projects, modelling and studying, modifying and extending code samples (Lye and Koh, 2014).

Metacognition through abstraction

Several frameworks have been suggested that support teachers, and their students, to build mental models of abstractions related to programming, which will help them to teach and learn how to program. Understanding abstraction can aid metacognition.

The Abstraction Transition Taxonomy (AT) divides student knowledge and practices in learning to program into three levels: code, computer science (CS) speak and English; AT also describes the transitions between these levels (Cutts et al, 2012). Another framework is Levels of Abstraction (LOA), a framework similar to AT but with four levels: execution of code, code, object and goals (Perrenet and Kaasenbrood, 2006). Armoni (2013) further developed this framework for high-school students, in which the 'object' level was renamed 'algorithm' level to support teacher and pupil understanding, and transitions across the levels were also defined. Statter and Armoni (2017) reported that learners using the framework showed improvements in attendance, algorithm development, algorithm creation, ability to explain solutions and understanding of initialization, with more improvement by girls than boys. To support primary classrooms, the LOA levels have been further renamed as running the code, code, design and task (Waite et al., 2018).

The importance of students being able to move from the 'task' level to the 'code' level and vice versa is linked with the advice that learners would benefit from being able to draw on existing templates or plans that solve a certain type of problem. To aid this, Lokkila et al. (2016) suggested that programmers would benefit from being taught 'learning templates' as a process for problem-solving.

Scaffolding

Scaffolding is used in education to describe both the micro-level scaffolding of teachers interacting with students in lessons and the macro-level scaffolding of planning lesson goals and the organization of learning tasks (Hammond and Gibbons, 2001). A simple example of a continuum of scaffolding has been developed to provide initial guidance for teachers to understand their choices better; the Computer Science Student-Centred Instructional Continuum (CS-SCIC) includes broad categories of instructional approaches for programming, such as copying code, targeted tasks, shared coding, project-based, inquiry-based and tinkering. This has been successfully used in England and the United States to support teachers' professional development (Waite and Liebe, 2021). CS-SCIC reflects the tension between exploration, making and direct teaching, and gives teachers a way to talk about planning programming lessons as they create a sequence of learning experiences (Waite and Liebe, 2021).

There are other ways to support learners that are discussed elsewhere in this book. Chapter 5 covers the development of problem-solving skills using the lens of computational thinking. Chapter 20 addresses programming misconceptions that teachers can usefully be aware of. Chapter 19 considers the use of planning and design in supporting learners to think algorithmically prior to getting stuck into code, and Chapter 14 focuses on the need for an understanding of appropriate language for programming. All these aspects add to our understanding of the effective teaching of programming in the classroom.

21.6 Conclusion

In this chapter, we have described research that teachers can use to support their teaching of programming in schools. The chapter has included a range of classroom strategies such as reading code and pair programming, contexts in which programming may be taught and how to support students. There is still much for teachers and researchers to learn from experience and robust research as our knowledge of this area continues to grow.

Key points

- Collaboration using strategies such as pair programming and peer instruction can promote both problem-solving and use of programming-specific language.
- Code comprehension activities facilitate an understanding of the program code, especially when mapped to the Block Model.
- Teacher modelling through live coding and worked examples enables learners to understand the process of programming.
- Programming can be taught in a variety of different contexts and environments, including physical computing and games and using unplugged approaches or block-based languages.
- Learners can be supported in their programming journey through cognitive apprenticeship, developing metacognitive skills and providing scaffolding

For further reflection

Reflect on the different needs of different learners: Using this chapter, make a mind map of the ways that you can support the learners in your classes who seem to have struggled learning to program, and elaborate on how that might be implemented in your classroom context.

Go deeper: Many of the ideas in this chapter are underpinned by fundamental theories of learning that can help us understand learning better. Here are some source texts that you may want to read to increase your understanding:

- Lave and Wenger (1991)
- Papert (1980)
- Savery and Duffy (1995)

References

Armoni, M. (2013), 'On Teaching Abstraction in Computer Science to Novices', *Journal of Computers in Mathematics and Science Teaching*, 32 (3), 265–84. https://www.learntechlib.org/p/41271.

Bau, D., Gray, J., Kelleher, C., Sheldon, J., and Turbak, F. (2017), 'Learnable Programming: Blocks and Beyond', *Communications of the ACM*, 60 (6): 72–80. https://doi.org/10.1145/3015455.

Bell, T., Alexander, J., Freeman, I., and Grimley, M. (2009), 'Computer Science Unplugged: School Students Doing Real Computing without Computers', *New Zealand Journal of Applied Computing and Information Technology*, 13 (1): 20–9. https://www.cosc.canterbury.ac.nz/tim.bell/cseducation/papers/Bell%20Alexander%20Freeman%20Grimley%202009%20JACIT.pdf.

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., Sharif, B., and Tamm, S. (2013), 'The use of code reading in teaching programming', in *Proceedings of the 13th Koli Calling international conference on computing education research*, 3–11. https://doi.org/10.1109/ICPC.2015.36.

Collins, A., Brown, J., and Newman, S. (1987), 'Cognitive Apprenticeship: Teaching the Craft of Reading, Writing, and Mathematics'. Technical Report No. 403. Center for the Study of Reading, University of Illinois at Urbana-Champaign. https://www.ideals.illinois.edu/bitstream/handle/2142/17958/ctrstreadtechrepv01987i00403_opt.pdf.

Cutts, Q., Esper, S., Fecho, M., Foster, S. R., and Simon, B. (2012), 'The Abstraction Transition Taxonomy: Developing Desired Learning Outcomes through the Lens of Situated Cognition', in *Proceedings of the 9th Annual International Conference on International Computing Education Research*, 63–70. https://doi.org/10.1145/2361276.2361290.

Denner, J., Green, E., and Campe, S. (2021), 'Learning to Program in Middle School: How Pair Programming Helps and Hinders Intrepid Exploration', *Journal of the Learning Sciences*. https://doi.org/10.1080/10508406.2021.1939028.

Denny, P., Luxton-Reilly, A., and Simon, B. (2008), 'Evaluating a New Exam Question: Parsons Problems', in *Proceedings of the Fourth International Workshop on Computing Education Research*, 113–24. https://dl.acm.org/doi/10.1145/1404520.1404532.

Ericson, B. J., Margulieux, L. E., and Rick, J. (2017), 'Solving Parson's Problems versus Fixing and Writing Code', in *Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, 20–9. https://doi.org/10.1145/3141880.3141895.

Garneli, V., Giannakos, M. N., and Chorianopoulos, K. (2015), 'Computing Education in K-12 Schools: A Review of the Literature', in *2015 IEEE Global Engineering Education Conference (EDUCON)*, 543–51. https://doi.org/10.1109/EDUCON.2015.7096023.

Hammond, J., and Gibbons, P. (2001), 'What Is Scaffolding?' in J. Hammond (ed.), *Scaffolding: Teaching and Learning in Language and Literacy education*, 1–14. Primary English Teaching Association. https://eric.ed.gov/?id=ED456447.

Hodges, S., Sentance, S., Finney, J., and Ball, T. (2020), 'Physical Computing: A Key Element of Modern Computer Science Education', *Computer*, 53 (4): 20–30. https://doi.org/10.1109/MC.2019.2935058.

Izu, C., Schulte, C., Aggarwal, A., Cutts, Q., Duran, R., Gutica, M., Heinemann, B., Kraemer, E., Lonati, V., Mirolo, C., and Weeda, R. (2019), 'Fostering Program Comprehension in Novice Programmers – Learning Activities and Learning Trajectories', in *Proceedings of the Working Group*

Reports on Innovation and Technology in Computer Science Education, 27–52. https://dl.acm.org/doi/10.1145/3344429.3372501.

Kafai, Y. B., and Burke, Q. (2015), 'Constructionist Gaming: Understanding the Benefits of Making Games for Learning', *Educational Psychologist*, 50 (4): 313–4. doi: 10.1080/00461520.2015.1124022.

Kafai, Y. B., and Resnick, M. (1996), *Constructionism in Practice: Designing, Thinking, and Learning in a Digital World*, Mahwah, NJ: Lawrence Erlbaum Associates.

Katai, Z., Toth, L., and Adorjani, A. K. (2014), 'Multi-Sensory Informatics Education', *Informatics in Education*, 13 (2): 225–40. https://www.learntechlib.org/p/158137/.

Lave, J., and Wenger, E. (1991), Situated Learning: Legitimate Peripheral Participation, Cambridge: Cambridge University Press.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Mostr, J. E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004), 'A Multi-National Study of Reading and Tracing Skills in Novice Programmers'. Working group reports from ITiCSE on innovation and technology in computer science education (ITiCSE-WGR '04), Association for Computing Machinery, New York, 119–50. https://dl.acm.org/doi/10.1145/1041624.1041673.

Lokkila, E., Rajala, T., Veerasamy, A., Enges-Pyykönen, P., Laakso, M., J., and Salakoski, T. (2016), 'How Students' Programming Process Differs from Experts—A Case Study with a Robot Programming Exercise', in *EDULEARN16 Proceedings of the 8th International Conference on Education and New Learning Technologies*, 1555–62. http://dx.doi.org/10.21125/edulearn.2016.1308.

Lye, S. Y., and Koh, J. H. L. (2014), 'Review on Teaching and Learning of Computational Thinking through Programming: What Is Next for K–12?' *Computers in Human Behavior*, 41: 51–61. https://doi.org/10.1016/j.chb.2014.09.012.

Margulieux, L. E., Morrison, B. B., Franke, B., and Ramilison, H. (2020), 'Effect of Implementing Subgoals in Code.org's Intro to Programming Unit in Computer Science Principles', *ACM Transactions on Computing Education*, 20 (4): 1–24. https://doi.org/10.1145/3415594.

Mayer, R. E. (2004), 'Should There Be a Three-Strikes Rule against Pure Discovery Learning?' *American Psychologist*, 59 (1): 14–19. https://doi.org/10.1037/0003-066x.59.1.14.

McDowell, C., Werner, L., Bullock, H., and Fernald, J. (2006), 'Pair Programming Improves Student Retention, Confidence, and Program Quality', *Communications of the ACM*, 49 (8): 90–5. https://doi.org/10.1145/1145287.1145293.

Papert, S. (1980), *Mindstorms: Children, Computers, and Powerful Ideas*, New York: Basic Books.

Parsons, D., and Haden, P. (2006), 'Parson's Programming Puzzles: A Fun and Effective Learning Tool for First Programming Courses', in *Proceedings of the Eighth Australasian Conference on Computing Education (ACE '06)*, 157–63.

Perrenet, J., and Kaasenbrood, E. (2006), 'Levels of Abstraction in Students' Understanding of the Concept of Algorithm: The Qualitative Perspective', *ACM SIGCSE Bulletin*, 38 (3): 270–4. https://doi.org/10.1145/1140124.1140196.

Qian, Y., and Lehman, J. (2017), 'Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review', *ACM Transactions on Computing Education*, 18: 1. https://doi.org/10.1145/3077618.

Repenning, A., Webb, D. C., Koh, K. H., Nickerson, H., Miller, S. B., Brand, C., Horses, I. H. M., Basawapatna, A., Gluck, F., Grover, R., Gutierrez, K., and Repenning, N. (2015), 'Scalable Game Design: A Strategy to Bring Systemic Computer Science Education to Schools through Game

Design and Simulation Creation', *ACM Transactions on Computing Education*, 15 (2): 1–31. https://doi.org/10.1145/2700517.

Robins, A., Rountree, J., and Rountree, N. (2003), 'Learning and Teaching Programming: A Review and Discussion', *Computer Science Education*, 13: 137–72. https://doi.org/10.1076/csed.13.2.137.14200.

Rubin, M. J. (2013), 'The Effectiveness of Live-Coding to Teach Introductory Programming', in *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 651–6. https://doi.org/10.1145/2445196.2445388.

Savery, J. R., and Duffy, T. M. (1995), 'Problem Based Learning: An Instructional Model and Its Constructivist Framework', Educational Technology, 35 (5): 31–8.

Schulte, C. (2008), 'Block Model: An Educational Model of Program Comprehension as a Tool for a Scholarly Approach to Teaching', in *Proceedings of the Fourth International Workshop on Computing Education Research (ICER '08)*, 149–60. https://doi.org/10.1145/1404520.1404535.

Sentance, S., Waite, J., and Kallia, M. (2019), 'Teaching Computer Programming with PRIMM: A Sociocultural Perspective', *Computer Science Education*, 29 (2–3): 136–76. doi: 10.1080/08993408.2019.1608781.

Shah, N., and Lewis, C. M. (2019), 'Amplifying and Attenuating Inequity in Collaborative Learning: Toward an Analytical Framework', *Cognition and Instruction*, 37 (4): 423–52. https://doi.org/10.1080/07370008.2019.1631825.

Statter, D., and Armoni, M. (2017), 'Learning Abstraction in Computer Science: A Gender Perspective', in *Proceedings of the 12th Workshop on Primary and Secondary Computing Education (WiPSCE'17)*, 5–14. https://doi.org/10.1145/3137065.3137081.

Sweller, J. (1994), 'Cognitive Load Theory, Learning Difficulty, and Instructional Design', *Learning and Instruction*, 4 (4): 295–312.

Teague, D., and Lister, R. (2014), 'Programming: Reading, Writing and Reversing', in *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 285–90. https://doi.org/10.1145/2591708.2591712.

Thomas, J. W. (2000), 'A Review of Research on Project-Based Learning'. Autodesk Foundation. https://www.asec.purdue.edu/lct/HBCU/documents/AReviewofResearchofProject-BasedLearning.pdf.

Tsai, C. Y., Yang, Y. F., and Chang, C. K. (2015), Cognitive Load Comparison of Traditional and Distributed Pair Programming on Visual Programming Language. 2015 International Conference of Educational Innovation through Technology (EITT), 143–6.

Vainio, V., and Sajaniemi, J. (2007), 'Factors in Novice Programmers' Poor Tracing Skills', in *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 236–40.

van Merriënboer, J. J. G., and Sweller, J. (2005), 'Cognitive Load Theory and Complex Learning: Recent Developments and Future Directions', *Educational Psychology Review*, 17 (2), 147–77. https://doi.org/10.1007/s10648-005-3951-0.

Venables, A., Tan, G., and Lister, R. (2009), '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*, 117–28. https://doi.org/10.1145/1584322.1584336.

Waite, J., Curzon, P., Marsh, D., Sentance, S., and Hawden-Bennett, A. (2018), 'Abstraction in Action: K-5 Teachers' Uses of Levels of Abstraction, Particularly the Design Level, in Teaching

Programming', *International Journal of Computer Science Education in Schools*, 2 (1): 14–40. https://doi.org/10.21585/ijcses.v2i1.23.

Waite, J., and Liebe, C. (2021), 'Computer Science Student-Centered Instructional Continuum', *in Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*, 1246. https://doi.org/10.1145/3408877.3439591/.

Weinman, N., Fox, A., and Hearst, M. A. (2021). 'Improving Instruction of Programming Patterns with Faded Parsons Problems', in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 53. https://doi.org/10.1145/3411764.3445228.

Weintrop, D., Killen, H., Munzar, T., and Franke, B. (2019), 'Block-Based Comprehension: Exploring and Explaining Student Outcomes from a Read-only Block-based Exam', in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*, 1218–24. https://doi.org/10.1145/3287324.3287348.

22

Epistemic Programming

Sven Hüsing, Carsten Schulte and Felix Winkelnkemper

Chapter outline

Chapter synopsis

This chapter introduces the idea of epistemic programming as a perspective on programming (projects) that focuses on its potential for gaining new insights, for example by analysing data. While epistemic programming is an exploratory (and typically data-driven) process, its core idea is to highlight the use of programming in terms of engaging in a subject of interest, of analysing it, of reflecting upon it and finally of using means of programming to present one's own thoughts, ideas and insights. A typical product of such a process is a computational essay: a mix of source code, visualizations and verbal explanations that presents ideas or insights gained through the (programming) process.

22.1 Introduction: Why should/would one engage in programming?

Within computer science education, it is a common goal to engage students in the art and craft of programming. Programming is often portrayed as being difficult, obscure and something for specialists, as described and questioned by Becker (2021). However, programming does not necessarily have to be seen as a burden. It can also be a passion and thereby a way of relaxing and of expressing oneself as it allows one to explore one's own world and/or fields of interest. To widen the perspective on programming towards an interest-driven programming process, we describe a didactic programming concept called epistemic programming (EP). Its focus lies on letting students explore their (local) world through programming while at the same time gaining insights into the programming process itself. The approach allows them to perceive programming as a means of (self-)expression, communication and knowledge acquisition. To illustrate the broad idea behind the approach, before even touching on the subject of programming, we first tell the story of an amateur photographer, who explores the world through taking pictures while at the same time learning about the technology behind photography.

One of the authors of this chapter considers himself to be a passionate photographer. He also happens to have an affection for London, where, naturally, he takes a lot of photos. Taking photos in London could be considered a completely useless activity. Virtually any landmark of the city has already been photographed numerous times, both by hobbyists and by professional photographers. One could therefore argue that nobody has to do so themselves any more, as 'professional solutions' already do exist. However, for a passionate photographer, taking photos in London remains to be rewarding for several reasons.

(1) Knowing existing photos of specific subjects which suggest it might be hard to photograph in a certain manner, it can be considered a challenge to try to take a similar photo, which means trying to figure out how the photo was created in terms of lighting, weather, lens, camera settings and post-processing. But being a hobby photographer is not only about trying to copy the work of others.

(2) One could also find satisfaction in trying to capture a subject in a different way than usual, which would, for example, mean finding new perspectives, creating creative compositions or trying to capture it in special lighting situations.

(3) Especially in very public places like in the City of London, a challenging and therefore rewarding undertaking might also be to try to find subjects which have indeed not been photographed before or which at least are not among the photographic clichés. This typically implies deviating from the usual paths a tourist would take. A hobby photographer, therefore, explores the location in one's own way by making the process of exploration itself one's own through taking photos. In this way, one expresses oneself through photography.

Being a passionate hobby photographer therefore means one is engaged in exploring two areas of interest at the same time. The first one is the location (or subject) one is taking photos of. While roaming around, trying to find interesting subjects or perspectives, waiting for certain situations to occur or considering different kinds of lighting, one explores even a common tourist hotspot quite differently than the average tourist. On the other hand, one also explores the technologies and concepts behind taking photos, when to use which kind of lens, how to set up the camera and how to process the photos afterwards. In exploring both the subject of photography and the architecture of photography, one is improving one's skills and knowledge in both areas. The outcome of this process is rewarding in itself. It is not just a copy of what others have done professionally but also a product of one's own creativity.

This little report of an enthusiast photographer indicates that photography can be way more than the activity of taking and collecting photos as it can be a means of exploring the world individually and expressing oneself. Moreover, it is about improving one's own skills and getting to know technical aspects about it. But what does this tale of photography as a leisure activity have in common with programming?

Similar to resorting to already existing, professionally created photos, one might argue that instead of oneself engaging in programming, one could merely use an already existing tool. While this may be an adequate argument for everyday problems, programming, as well as photography, can also be about expressing oneself through the individuality of the programming process itself as well as through its products. When taking photos while on holiday for the purpose of sending them to friends or family, these photos are supposed to transport a certain message. According to the expression/self-development-dimension described by Schulte (2013: 21), the same can be said about programming, as 'programs unavoidably include ideas or problem solutions so that "others can reason about"' it: in this characterization, programming is a way to express oneself through the creation of an aesthetic product. With programming, one could, for example, create a digital greeting card for one's own family. In this case, one would also follow a certain idea of expressing oneself, which would have a quite aesthetic character.

In addition to that, EP may also serve the goals of generating ideas and searching and finding evidence for certain theories and ideas. Kay (2005), for example, describes how children could program a simulation of falling objects with different weights to show others the weight-depending behaviour of these objects. Programming could similarly be used in order to evaluate the air quality in the classroom. In this case, students would focus on gaining and communicating insights (about air quality) through their programming process. As, in these examples, the goal of getting information about one's world is rather more factual than in the previous, more artistic examples, this category of EP aligns particularly well with data analysis and data science/AI in general. However, even then, the aesthetic character remains an important factor, especially when transferring one's findings to other people.

These examples of EP can be related to the heuristic or tinkering approach in which they are performed: when creating a greeting card, when developing a simulation and when evaluating data, one repeatedly switches between phases of adapting code and running it in order to discover areas where improvements may still be needed (e.g. if the greeting card is not yet to one's liking, the simulation is not yet running accurately, or the appropriate evidence for a hypothesis has not yet

been found). It is the same kind of short feedback loop which can be observed in digital photography. Here, one would tinker with different camera angles, various zoom levels, different camera settings or lighting arrangements; check the results; adapt settings when necessary and try again.

Another similarity between photography and programming is that professional execution is quite complicated and complex. In photography, there are many aspects that need to be considered and specific 'photography skills' that are to be used and have to be learned in advance. Nevertheless, most of us would all agree to be some kind of 'hobby-photographer', at least in certain situations. Similarly, programming, while being an undertaking which can be carried out with high levels of skills, can also be performed and therefore be targeted at non-experts, who could use it as a tool for their own purpose, for example to find out something about an area they are interested in.

Just like you do not need to study photography to take a few nice shots and explore the world (while simultaneously learning something about the camera as well as the techniques of photography), leisure or – as they are sometimes called – end-user programmers (Ko et al., 2011) can also make active use of computers to explore the world, the technology and themselves. Through such an insight-driven programming approach, a computer can serve as a sort of magnifying glass on the world, so that one's own environment can be explored in a way which would not have been possible without the technical aid of programming.

As programming can be seen as a tool to both express oneself and to perceive the world, mastering it to an adequate degree can be regarded as a kind of literacy, just like reading and writing. As Vee puts it: 'It is important to widen access to programming because of its power and diversity of applications, which means that programming cannot be relegated to the exclusive domain of computer science. It is also important to broaden our concepts of writing to include programming. Together, images, sound and other modes of composition have already shifted the way we communicate and how we can express and process information' (Vee, 2013: 60). Programming allows many kinds of compositions. In addition to fostering the exploration of a domain one is interested in, programming opens up and is an essential part of expressing ideas and communicating them.

With these goals in mind, EP aims at creating a specific type of interactive document, often called a computational essay. A computational essay is a 'genre of scientific writing that combines live code, written prose, mathematics, and pictures or diagrams in order to make an argument, explain an idea, or tell a story' (Odden and Malthe-Sørenssen, 2021: 15704). One could argue that the programming code itself already is a representation of the programming process, which can be further elaborated by additional explanations of the code as well as the programming results. In EP, we therefore explicitly introduce computational essays to enable students to keep track of their programming processes and the processes of creating self-expressions, ideas or evidence.

22.2 Characterizing epistemic programming

EP focuses on the acquisition of new insights and the expression and representation of ideas. The word 'epistemic' comes from the Greek, meaning 'knowledge'. Bereiter (1980: 88) introduced 'epistemic writing' to describe a way to gain knowledge through the act of writing, which combines the 'skill system for reflective thought' and the 'skill system for unified writing', to search for

meaning within a context. Writing, in this interpretation, becomes an essential part of thinking. Others conclude that the epistemic character of writing can also be experienced by novice writers or younger students in general (Casa et al., 2016; Galbraith, 1999; Pohl and Steinhoff, 2010; Strohmaier, Vogel and Reiss, 2018). Keys (1999) mentions the idea within different scientific fields to reflect on the respective content as well as to produce new knowledge through the act of writing itself. An important aspect of writing is that it can help to present knowledge explicitly in order to find connections or to think intensively about something based on the self-produced representation (Keys, 1999). Just like writing text, programming can represent an activity through which insights can be gained, be made explicit and be structured. Therefore, we call this kind of programming epistemic programming. To summarize, the goals behind the EP approach are as follows:

(1) To convey that programming can be a tool to gain knowledge: students should be able to reflect that programming empowers the programmer to gain new insights. The programming process, therefore, is synchronized with the knowledge-acquisition process in terms of that hermeneutic, slowly advancing, tinkering process of digging deeper and deeper into a specific topic.
(2) To dissolve the prejudices of the 'nerdiness' of programming by showing that it can play a major role in everyday life as (epistemic) programming represents a tool to discover one's own fields of interest in one's own world.
(3) To include both the understanding of the (local) environment (external insights) and the acquisition of competent handling and understanding of digital artefacts for this very purpose (internal insights) (see the programming dimension of thinking in Schulte (2013)).
(4) To communicate the intertwined knowledge acquisition and programming processes: Representing and recording expressions, ideas and evidence together with the respective programming process in order to reflect on these aspects and to distribute them to others.

EP projects can be of different grades of complexity: On the one side, one could focus on complex/scientific research projects, where there is a concrete question that shall be answered. On the other side, one could just casually tinker with the means of programming to explore one's own world without a specific epistemic question in mind or even without any desire for producing new insights at all.

In the next section, we will explore the concept of EP in more depth.

22.3 Epistemic programming projects

To further clarify the concept of EP, we use the 4P's (people, project, product and process) often used to characterize software development (Jacobson, Booch and Rumbaugh, 1999). By doing so, we address the following questions:

- People: Who is EP aimed at; what roles do people have?
- Project: What sort of projects can be performed using an EP approach?
- Process: How can EP processes take place?
- Product: What characterizes the product of EP?

People

EP aims at making programming appeal to a wider audience: In a broad sense, even musicians using programming for composing and performing music engage in EP-like activities.[1] In a similar direction, EP can be identified within the idea of storytelling with programming, often focusing on children using block-based languages, which are then used as a didactical tool for introducing 'real' programming (e.g. Adams and Webster, 2012; Šnajder, 2014). In a way similar to our introductory example, a study with graphic designers and photographers states that they were engaging in programming as scripting tool features in GIMP or Adobe Photoshop, mostly to save time, create custom effects or share artefacts (Dorn and Guzdial, 2006).

A major group for whom EP is useful or even mandatory are scientifically minded people – putting emphasis on the potential of EP as an activity to gain insights and evidence. As Zwart (2018: 979) describes, code 'is a description of concepts and their relationships, which are imperative for reproducibility and validating the [scientific] results'; thus, without the ability to properly use this language, scientific progress is hindered (Zwart, 2018). Accordingly, EP can be seen as the skill needed by scientists in general for analysing and interpreting data and presenting and communicating their results (Somers, 2018). Again, also youth (i.e. laypersons) can use this to produce data-driven stories (Wilkerson et al., 2021). In this respect, EP can be classified as part of the scientific tradition of programming (see Chapter 2 in this book), as it is an approach of going through science-like cognitive processes.

Project

Having identified a wide range of people potentially being involved in EP, a similarly wide range of projects can benefit from it: Storytelling projects are projects where EP is used to develop, represent and share ideas. Programmers communicate them through an artefact that also has aesthetic intentions. These involve writing (e.g. in Twigg, Blair and Winter, 2019), music, graphic design or photography. These projects are likely to be open and exploratory while producing an interactive medium. They can be performed individually or in a group effort.

Data-driven stories are a specific form of storytelling, for example used by data journalists. Here, the role of data and of programming to explore and analyse data becomes much more prominent. Data stories as projects are often driven by some 'epistemic need' (see 'general epistemic need' in Kidron et al. (2010)) such as a question that might be answerable by evaluating data. Following this epistemic need, students might progress in a typical pattern of data science projects (see, e.g., Pfannkuch et al., 2010; Wild and Pfannkuch, 1999).

Data science projects could be called the most strict and scientific form of EP projects. They strive for evidence and represent it in a form that allows criticism and scrutiny by others. Especially by not only telling a data story but also including the code used to manage, clean, analyse and visualize data together with additional descriptions, they assure reproducibility. EP projects, therefore, allow others to also follow the process of enquiry, to reproduce it as well as to test it by changing the code and hence experiment with variations of the scientific analysis.

Process

Within the range from open storytelling projects via data-driven storytelling to (professional) data science projects, the process of programming becomes more and more explicit, controlled and reproducible. Nevertheless, an important characteristic of EP remains to be its inherent tinkering style. Dong et al. (2019) identify tinkering processes in the field of open-ended assignments, allowing more opportunities for creative solutions and/or to pursue a set of different goals; it is becoming observable by back-and-forth changes in the code and is in this view connected to uncertainty, hesitation and indecisiveness (Dong et al., 2019: 1206). The article concludes that any of the observed tinkering behaviours can be either productive or detrimental. EP stresses the positive connotations of tinkering, interpreting it in line with using programming and coding like epistemic writing as a tool for thinking, similar to what Dong et al. (2019) call prototype-based tinkering. All in all, the key aspect of this process is the cognitive (and probably also motivational) function of building a representation of one's idea that helps to clarify the idea and one's own thinking.

In the more typical view on programming in computer science education, programming is more aligned to systematic problem solving and implementing algorithms. Coding is therefore always dependent on a previous modelling and design activity or on features of tasks that pre-structure the solution (e.g. to implement a certain algorithm). From this perspective, tinkering would rather be seen as a sign of a lack of competencies, an obstacle in reaching one's goals and a habit one should overcome. In contrast, in EP, tinkering is merely regarded as an indication that the process of thinking and gaining insights is becoming iteratively more and more profound. The process is targeted towards developing ideas and stories, creating meaning, getting insights and/or producing evidence – just as in writing, where experts plan, write and especially revise what they have written. In this process, one switches one's role between being a writer and critic of oneself. The (intermediate) code is an interim representation of the current ideas and thought processes and is used to elaborate and refine one's own thinking as well as ideas and insights.

EP can be implemented on various levels of complexity and could even serve the purpose of an introduction to programming. It can be aligned to ideas of data science projects as outlined in, for example, Ridsdale et al. (2015) or Wild and Pfannkuch (1999). In this context, EP affords and communicates aspects of programming that otherwise might be unrepresented. Following the EP approach, such aspects get the attention they deserve, not only as a prerequisite for writing computational essays but also as part of some everyday programming literacy. To hint at some of these aspects:

a) In data science process models, for example, data has to be represented and therefore cleaned and probably transformed.
b) Many aspects like analysing data, visualizing data as well as data transformations are supported by libraries. Their use and the ability to familiarize oneself with a library is essential.
c) In many projects, programming is maybe closer to scripting, which according to Loui (2008) means relying on high-level abstractions and provides means for rapid prototyping, working with data in heterogeneous settings and getting things done with short source code snippets. It can also be interpreted as glueing together powerful components (Ousterhout, 1998).

Generally, this tinkering or scripting process within EP projects typically proceeds in a kind of spiral-like process: As the programming process is driven by new insights, the findings of each iteration of code-adaption need to be reflected upon in order to optimize the program according to the own research interest. This results in a circular sequence of *code adaption*, *code execution*, *exploration of results* and *reflection of the results*. From iteration to iteration, the code is therefore adapted to gradually gain more and more concrete insights, emerging from the programming results.

Product

There are numerous tools which would support or allow such tinkering processes covering numerous areas of interest – often making them accessible by relying on the direct manipulation principle. Just like in the photography example, one could argue that these could just be used without the need of programming something oneself. However, an important advantage of programming over using such tools is that the process is captured in the program code itself and can therefore be revisited at a later time. In this regard, EP and creating computational essays have the goal of making this thinking process permanent and thereby comprehensible to the reader of the program code, so that after the program code is written, one can understand this process, reflect on it, reproduce it or even further develop it. A specific aspect of this exploratory and/or tinkering process is that it blurs the distinction between (a) planning or designing, (b) implementing and (c) testing. All three are part of the stepwise creation of knowledge or insights, hence the product of this process.

The product of an EP project is not only the program itself but also the insights and the way they were gained, as the program typically does not fully convey its process of creation. Computational essays (DiSessa, 2000; Odden and Malthe-Sørenssen, 2021; Wolfram, 2017) can fill this gap. Within this medium, it is possible to combine the aspects of program code, programming execution as well as the documentation of the program code and the interpretation of its results. By creating a computational essay as an explicit product, it is possible to make the knowledge acquisition process as well as the programming process accessible to the 'readers', thereby also empowering them to tinker with the program for validating the results or gaining their own insights (as McNamara (2019) and Sandve et al. (2013) claim), allowing them to evolve into 'writers'.

To create computational essays, Jupyter Notebooks (see Figure 22.1) are a suitable tool, as through their block structure, they allow programmers to record the programming process as well as the recognition or thought processes (Granger and Perez, 2021; Johnson, 2020). Program code can be segmented across different code cells, which can then be executed separately. The output of each code cell is displayed just below the respective cell. So-called markdown cells are used to comment on the programming code and the programming results. The programming and the knowledge acquisition process can be explained step by step through documenting the program code and interpreting the programming outputs within markdown cells. The final computational essay often needs cleaning and polishing of the versions produced during tinkering/scripting and commenting/documenting of ideas and results (see Rule, Tabard and Hollan (2018) for further discussion).

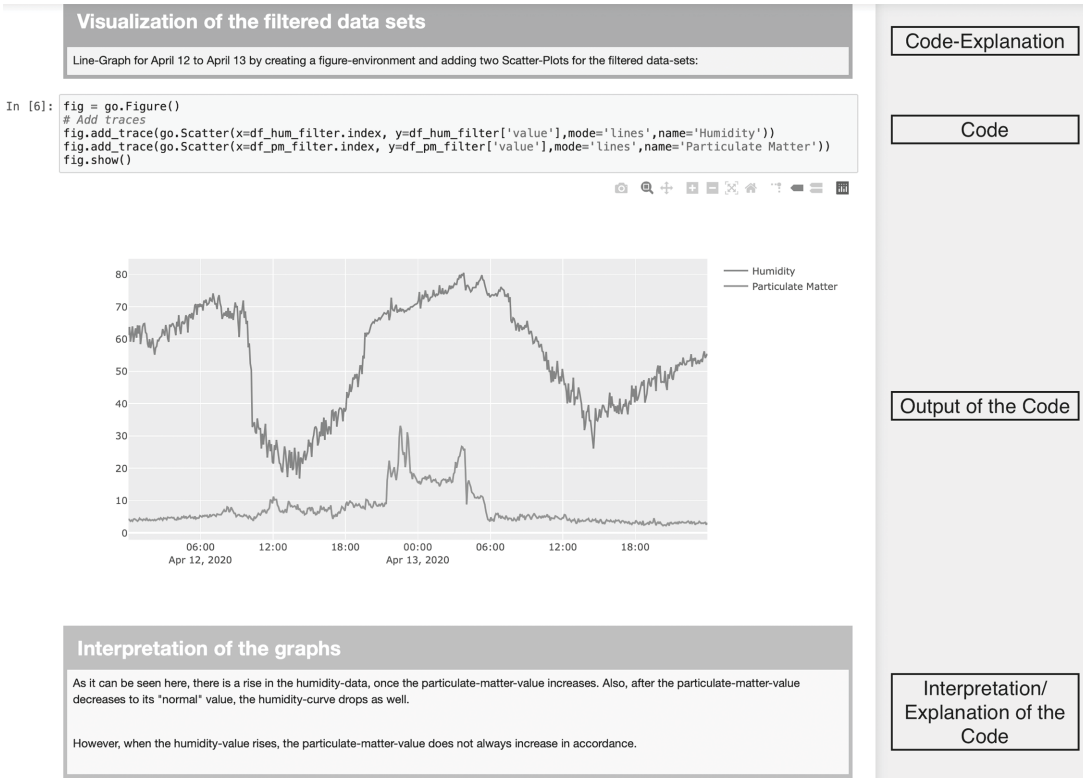

Figure 22.1 Elements of a computational essay in a Jupyter Notebook

Key concept: Peer instruction

EP is characterized by the main goals of gaining insights, forming and representing ideas, and communicating:

- The *product* of EP is a computational essay that integrates program code as well as natural text in one place, thereby representing an executable and adaptable knowledge product, which aims to present and explain some insights or ideas.
- The *process* of EP is often based on tinkering, developing, testing and refining hypotheses where program code is used to represent ideas as well as to test and document them.
- The *people* addressed by EP are not only professional programmers but also laypersons who can use it to explore their own projects and interests and to be able to participate in a new form of communication (think, e.g., about data journalists).
- The *projects* typically carried out as EP projects are often based on data to be explored or on some phenomenon in one's own sphere of interest.

22.4 Exemplifying the main aspects of epistemic programming

Within EP, students are supposed to learn how to use digital artefacts and (quite likely) data within these artefacts to create meaningful programs or evaluations and therefore to gain specific insights into the environment from which the data was obtained. Especially in the context of data science, EP provides a different perspective on programming by showing the students how the world can be made perceivable through data and how the model based on the data can be made explorable through programming. Programming appears as a tool to create insights, to be able to open up the world and thus to act self-determined and autonomously in society. This illustrates the educational relevance of EP (see the term 'Bildung' in Hopmann (2007) and Schulte and Budde (2018)).

A currently relevant example for insight-driven programming in a school context is the analysis of environmental data – an EP project that has already been tested within different school classes (Hüsing and Podworny, 2022). In this project, students engage in measuring and analysing different values, for example particulate matter, CO_2, humidity or temperature, in the surrounding of their school to compare it to other cities or for different locations around their school (e.g. near a street and in a park). Here, programming is a suitable tool to gain and share such insights, for example to raise awareness for environmental protection on a local example. Within the project, students choose their own research focus for the data analysis and then collect suitable data, before analysing it. The whole process is constantly driven by an individual epistemic interest that the students are working on. The students try different evaluation methods and possibly also different data sets to create meaningful visualisations and evaluations in order to find an answer to their research question. When doing so, the students' research focus is likely to be subject to change based on the analysis and its results, as new insights might also lead to new questions or interests. Finally, the students share their findings as well as their programming and knowledge acquisition process in a computational essay. In total, the focus is not only on the software product but more importantly also on the programming- and knowledge-acquisition process itself.

Key points

- Programming is not only a necessity in order to build new software products but also a tool for thinking, gaining and presenting (new) insights.
- Epistemic programming will be an important literacy for communication in a digital and data-driven world.
- Epistemic programming allows a different introduction to programming and programming languages that might spark interest and motivation in such learners who would not be interested in learning programming per se.
- Epistemic programming allows a low threshold to include data science in the computer science classroom.

- Epistemic programming is not solely focused on teaching job-related skills but (like the photographer mentioned at the beginning of the chapter) to foster subjectivation and developing individual capacities to pursue one's own interests and curiosity.

All in all, there are two main perspectives from which to look at this approach: from a perspective of computer science education, EP might represent a rather different and promising approach to inspire students to be interested in programming. It therefore could also change the students' perception of programming in terms of relevant applications of programming skills (a).

From a knowledge-based view, students can gain individual insights through programming – on the one hand, about the digital artefacts (including the use of data) (internal insights) and, on the other hand, into a specific domain they are interested in (external insights) (see the thinking dimension of programming in Schulte (2013)). Programming enables them to gain insights that they could not have been able to get without it (b).

In the previous example of an environmental analysis, the insight-driven characteristic of EP becomes quite clear: While in other school-based programming approaches, the focus is often on products in the sense of software, on implementing an algorithm and on the students' development of specific programming skills, within EP projects, students develop their own insight-driven focus and align their programming process with it.

Because of these intrinsic characteristics of EP, it might be a valuable approach to introduce the students to that kind of 'leisure', 'follow-your-interest' characteristic as well as a technique through which students can develop specific ideas or evidence and communicate them to others, thereby learning something about using data and digital artefacts and at the same time something within the context of their own interest.

For further reflection

- Reflect on your own programming projects, to what extent can they be interpreted as epistemic programming projects.
- Read the paper 'The State of the Art in End-User Software Engineering' by Ko et al. (2011) and compare the epistemic programming approach to end-user programming and end-user software development.
- Chapter 18 presents didactic principles for teaching programming at school: Do you think the principles mentioned there are also applicable and essential for teaching epistemic programming?
- Epistemic programming depends on the interaction with some kind of digital artefact, like Jupyter Notebooks. Use the ARIadne principle, described in Chapter 4, to analyse such artefacts in the context of interaction within the epistemic programming process.

Note

1 See, for example, Blackwell and Collins (2005) with a discussion of programming languages for live coding and Brown and Sorensen (2009: 18) from a more compositional perspective on new practices and interaction with generating music, addressing the 'live coder' for whom 'software code is a medium of expression by which creative ideas are articulated'.

References

Adams, J. C., and Webster, A. R. (2012), 'What Do Students Learn about Programming from Game, Music Video, and Storytelling Projects?' in *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, 643–8. https://doi.org/10.1145/2157136.2157319.

Becker, B. A. (2021), 'What Does Saying That "Programming Is Hard" Really Say, and about Whom?' *Communications of the ACM*, 64 (8): 27–9. https://doi.org/10.1145/3469115.

Bereiter, C. (1980), 'Development in Writing', in L. W. Gregg and E. R. Steinberg (eds), *Cognitive Processes in Writing*, 73–93, Hillside, NJ: Erlbaum.

Blackwell, A., and Collins, N. (2005), 'The Programming Language as a Musical Instrument', in P. Romero, J. Good, E. Acosta Chaparro and S. Bryant (eds), in *Proceedings of PPIG 17*, 11.

Brown, A. R., and Sorensen, A. (2009), 'Interacting with Generative Music through Live Coding', *Contemporary Music Review*, 28 (1): 17–29. https://doi.org/10.1080/07494460802663991.

Casa, T., Firmender, J., Cahill, J., Cardetti, F., Choppin, J., Cohen, J., Cole, S., Colonnese, M., Copley, J., DiCicco, M., Dieckmann, J., Dorl, J., Gavin, M., Hebert, M., Karp, K., LaBella, E., Moschkovich, J., Moylan, K., Olinghouse, N., and Zawodniak, R. (2016), Types of and Purposes for Elementary Mathematical Writing: Task Force Recommendations. http://mathwriting.education.uconn.edu.

DiSessa, A. A. (2000), *Changing Minds: Computers, Learning, and Literacy*, Cambridge, MA: MIT Press.

Dong, Y., Marwan, S., Catete, V., Price, T., and Barnes, T. (2019), 'Defining Tinkering Behavior in Open-Ended Block-Based Programming Assignments', in *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 1204–10. https://doi.org/10.1145/3287324.3287437.

Dorn, B., and Guzdial, M. (2006), 'Graphic Designers Who Program as Informal Computer Science Learners', in *Proceedings of the Second International Workshop on Computing Education Research*, 127–34. https://doi.org/10.1145/1151588.1151608.

Galbraith, D. (1999), 'Writing as a Knowledge-Constituting Process: Knowing What to Write', *Conceptual Process in Text Production*, Amsterdam: Amsterdam University Press, 139–64.

Granger, B. E., and Perez, F. (2021), 'Jupyter: Thinking and Storytelling with Code and Data', *Computing in Science & Engineering*, 23 (2): 7–14. https://doi.org/10.1109/MCSE.2021.3059263.

Hopmann, S. (2007), 'Restrained Teaching: The Common Core of Didaktik', *European Educational Research Journal*, 6 (2): 109–24. https://doi.org/10.2304/eerj.2007.6.2.109.

Hüsing, S., and Podworny, S. (2022), 'Computational Essays as an Approach for Reproducible Data Analysis in lower Secondary School', in *Proceedings of the IASE 2021 Satellite Conference*. https://doi.org/10.52041/iase.zwwoh

Jacobson, I., Booch, G., and Rumbaugh, J. (1999), *The Unified Software Development Process*, Reading, MA: Addison-Wesley.

Johnson, J. W. (2020), 'Benefits and Pitfalls of Jupyter Notebooks in the Classroom', *Data Science*, 6, 32–37.

Kay, A. (2005), Squeak Etoys, Children & Learning. Viewpoints Research Institute Research Note. http://www.vpri.org/pdf/rn2005001_learning.pdf.

Keys, C. W. (1999), 'Revitalizing Instruction in Scientific Genres: Connecting Knowledge Production with Writing to Learn in Science', *Science Education*, 83 (2): 115–30. https://doi.org/10.1002/(SICI)1098-237X(199903)83:2<115::AID-SCE2>3.0.CO;2-Q.

Kidron, I., Bikner-Ahsbahs, A., Cramer, J., Dreyfus, T., and Gilboa, N. (2010), 'Construction of Knowledge: Need and Interest', in *Proceedings of the 34th Conference of the International Group for the Psychology of Mathematics Education*, 3: 169–76.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., and Wiedenbeck, S. (2011), 'The State of the Art in End-User Software Engineering', *ACM Computing Surveys*, 43 (3): 1–21. https://doi.org/10.1145/1922649.1922658.

Loui, R. P. (2008), 'In Praise of Scripting: Real Programming Pragmatism', *Computer*, 41 (7): 22–6. https://doi.org/10.1109/MC.2008.228.

McNamara, A. (2019), 'Key Attributes of a Modern Statistical Computing Tool', *American Statistician*, 73 (4): 375–84. https://doi.org/10/ghnfwp.

Odden, T. O. B., and Malthe-Sørenssen, A. (2021), 'Using Computational Essays to Scaffold Professional Physics Practice', *European Journal of Physics*, 42 (1): 015701. https://doi.org/10/gjf7nq.

Ousterhout, J. K. (1998), 'Scripting: Higher Level Programming for the 21st Century', *Computer*, 31 (3): 23–30. https://doi.org/10.1109/2.660187.

Pfannkuch, M., Regan, M., Wild, C., and Horton, N. J. (2010), 'Telling Data Stories: Essential Dialogues for Comparative Reasoning', *Journal of Statistics Education*, 18 (1): 9. https://doi.org/10.1080/10691898.2010.11889479.

Pohl, T., and Steinhoff, T. (eds) (2010), *Textformen als Lernformen*, vol. 7, 5–26, Duisburg: Gilles und Francke Verlag. https://kups.ub.uni-koeln.de/8220/.

Ridsdale, C., Rothwell, J., Smit, M., Ali-Hassan, H., Bliemel, M., Irvine, D., Kelley, D., Matwin, S., and Wuetherick, B. (2015), 'Strategies and Best Practices for Data Literacy Education: Knowledge Synthesis Report', Dalhousie University. http://dalspace.library.dal.ca/bitstream/handle/10222/64578/Strategies%20and%20Best%20Practices%20for%20Data%20Literacy%20Education.pdf?sequence=1.

Rule, A., Tabard, A., and Hollan, J. D. (2018), 'Exploration and Explanation in Computational Notebooks', in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 1–12. https://doi.org/10/gfw4vk.

Sandve, G. K., Nekrutenko, A., Taylor, J., and Hovig, E. (2013), 'Ten Simple Rules for Reproducible Computational Research', *PLoS Computational Biology*, 9 (10): e1003285. https://doi.org/10/pjb.

Schulte, C. (2013), 'Reflections on the Role of Programming in Primary and Secondary Computing Education', in *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*, 17–24. https://doi.org/10.1145/2532748.2532754.

Schulte, C., and Budde, L. (2018), 'A Framework for Computing Education: Hybrid Interaction System: The Need for a Bigger Picture in Computing Education', in *Proceedings of the 18th Koli Calling International Conference on Computing Education Research,* 10.

Šnajder, L. (2014), 'Scratch as a Glue for Funny Programming, Curiosity and Music Creation', *Programme Committee*, 228–36. https://core.ac.uk/download/pdf/33693352.pdf#page=228.

Somers, J. (2018), 'The Scientific Paper Is Obsolete. Here's What's Next—The Atlantic', *Atlantic*, 12.

Strohmaier, A., Vogel, F., and Reiss, K. M. (2018), 'Collaborative Epistemic Writing and Writing-to-Learn in Mathematics: Can It Foster Mathematical Argumentation Competence?' *RISTAL*, 1 (1): 135–49. https://doi.org/10/gmvbmq.

Twigg, S., Blair, L., and Winter, E. (2019), 'Using Children's Literature to Introduce Computing Principles and Concepts in Primary Schools: Work in Progress', in *Proceedings of the 14th Workshop in Primary and Secondary Computing Education*, 1–4, Glasgow, Scotland: ACM.

Vee, A. (2013), 'Understanding Computer Programming as a Literacy', *Literacy in Composition Studies*, 1 (2): 42–64. https://doi.org/10.21623/1.1.2.4.

Wild, C. J., and Pfannkuch, M. (1999), 'Statistical Thinking in Empirical Enquiry', *International Statistical Review*, 67 (3): 223–48. https://doi.org/10/fdhtmj.

Wilkerson, M., Finzer, W., Erickson, T., and Hernandez, D. (2021), 'Reflective Data Storytelling for Youth: The CODAP Story Builder', *Interaction Design and Children*, 503–7. https://doi.org/10.1145/3459990.3465177.

Wolfram, S. (2017), 'What Is a Computational Essay?' Stephan Wolfram Writings. https://writings.stephenwolfram.com/2017/11/what-is-a-computational-essay/.

Zwart, S. P. (2018), 'Computational Astrophysics for the Future', *Science*, 361 (6406): 979–80. https://doi.org/10.1126/science.aau3206.

Glossary

Algorithm	A process for solving a problem or achieving an outcome, built from steps that a computational device can execute.
ARIadne principle	A way of analysing digital artefacts by describing their architecture and their relevant societal discourses and interrelating them by retracing their genesis and by focusing on the interactions relevant stakeholders have with the artefact.
Binary representation	Representing information on a computer using just two digits (usually written as 0 and 1).
Coding	A loose term for computer programming (and when used more precisely, the part of programming that involves converting an algorithm or plan to a programming language); the word has other common meanings in computer science, including encryption coding, compression coding, channel coding, binary codes, markup language codes and more.
Competency	A complex disposal of behaviour that can be applied to solve a certain task or problem that is relevant in "real" life.
Computational thinking	A way of thinking about a problem and the important elements of it and breaking the problem down into steps that can be implemented on a computer.
Constructionism	A form of constructivism that emphasizes student learning through the creation of concrete projects that are shared with others; constructionism has inspired the design of toolkits such as Logo, Lego Mindstorms and Scratch.
Constructivism	A theory of learning that describes how students start to build their knowledge and understand though questioning, direct experience and reflection.

Culturally responsive teaching	The inclusion of students' cultural references as a way to empower students to learn.
Curriculum	Characterization of learning in terms of rationale, content, learning outcomes and instructional strategies; a curriculum can take the form of a formal document ('intended' curriculum) but can also appear as the learning that actually takes place ('implemented' or 'attained' curriculum).
Debugging	Working out why a program doesn't do what it was intended to, and fixing the problem.
Design	Working out useful details of what is to be implemented as a program from the task or problem description (the requirements); the design is likely to include components such as the algorithm, a data structure design and graphical user interface design; for younger learners these components can be simplified, for example as the algorithm, the object and data design, the artwork and sound design.
Diversity	The representation of different kinds of individuals and different kinds of social or cultural groups.
Duality reconstruction	The analysis of digital artefacts in terms of architecture ('how does it work') and relevance ('what does it do') in order to understand the interaction between the artefact and the outside world.
Educational standard	A set of competencies depicted in detail that were decided by educational authorities to be the minimal or average learning outcomes of educational institutions.
Equity	The creation of opportunities for historically underrepresented populations to have equal access to and participation in computer science education.
Growth mindset	The belief that one's abilities can be developed through hard work.
Hybrid network	A network consisting of human and digital actors, with human–human, human–computer and computer–computer connections and interactions.
Inclusion	The active and intentional engagement with diversity such that a range of individuals are able to fully participate.
Integrated learning	Learning where multiple subjects are used at the same time.

Interaction	The interplay between digital artefacts and the world, which can be viewed on three levels: interaction between a human and a computer, interaction within more complex hybrid networks and interaction between computing and society as a whole.
Intersectionality	The interconnected nature of social categorization (e.g. gender, culture, ability).
Kinaesthetic activities	Activities where physical, tangible objects are used as representatives of abstract concepts and/or where physical activities are used.
Learning objective	A description of goals that educators aim to achieve in terms of learning progress of their students.
Meta-design	Design of digital artefacts as a (socio-technical) problem solving framework that allows others to design their solutions, as opposed to design of such artefacts by directly implementing a concrete problem-solving strategy.
Metonymy	A figure of speech in which a thing is not mentioned directly but via a closely associated thing (e.g. the name Hollywood can be used to refer to the movie industry associated with that neighborhood); extremely common in human language but not in programming languages.
Misconception	An underdeveloped or flawed idea about specific content; different educational theories have different definitions for what a misconception (or 'alternative conception') is; we use the term in a loose sense.
Notional machine	The capabilities of a particular programming system (language and environment), which the system draws on as it executes programs; 'What the system can do for the programmer, and what it can't'; understanding a notional machine is necessary in order to reason reliably about program behaviour and to instruct the system effectively.
Objectivation	The identification of (in particular, human) actors and their behaviour with information processing units and processes.
Pair programming	Two people work together at one computer to write code, switching frequently between working the keyboard and mouse and directing the work.

Pedagogical content knowledge	Knowledge about particular content from the perspective of learning and teaching: what difficulties does the content pose for learners, what methods and tools work for teaching the content and so on.
Semantic waves	An approach to explanation that involves giving technical concepts but then relating them in some way to concrete (or material) situations or contexts that are already understood, before then explicitly linking back to the new concept.
Superbug	The beginner mistake of instructing a computer much as one would instruct a human being with interpretive powers.
Translanguaging	Designing instruction where students can access multiple languages to maximize communication.
Turing completeness	A distillation of the kinds of structures that digital devices can execute, which helps to understand the limits of computation and the range of structures needed to write programs.
Turtle-based language	A programming language based on movement (forward, left, right, etc.), either physically or on a screen.
Universal design for learning	A framework to improve learning through flexible design that can meet the needs of students with diverse learning needs.
Unplugged	Teaching computer science concepts without using a computer, or at least, without using programming as the vehicle.

Index