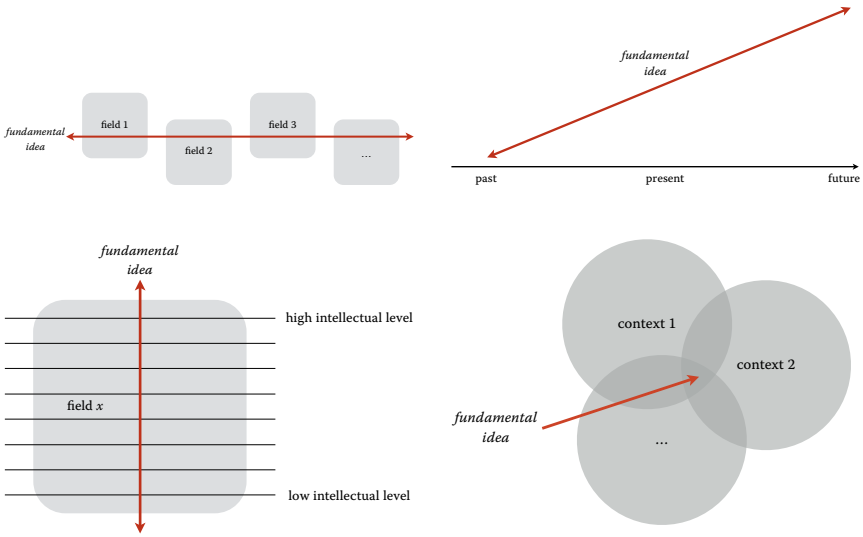
Characterising fundamental ideas in international computer science curricula

Author
Tim Steenvoorden, BSc

Supervisor
prof. dr. Erik Barendsen

Second Assessor
dr. Sjaak Smetsers



Department of Computer Science
Faculty of Science
Radboud University Nijmegen
August 2015

Abstract

Computer science as a subject in high schools is a hot topic. Many countries are looking for methods to teach computing principles to children as soon as possible. Finding a good way to do this is a difficult task which every country solves in its own way. In this thesis we present tree main contributions to computer science education at (junior) high schools.

Firstly we show a quantitative and qualitative *comparison* of the computer science curricula and guidelines from France, England, the United States and the Netherlands. Using categorisation of concepts in fields of the discipline, we analyse the content knowledge of each document.

Secondly we exhibit a different angle towards computer science curriculum contents based on the notion of *fundamental ideas*. A curriculum based on fundamental ideas makes sure students can better relate topics and create a strongly connected cognitive structure on the subject.

Thirdly we develop a *conceptual framework*, based on fundamental ideas, to embed computer science education within STEM. Curricula based on this framework make sure to fit well into (junior) high school STEM education and incorporate all fundamental ideas of computer science. The process towards such a framework uncovers which fundamental ideas in computer science are similar to STEM practices and principles and which are specific for the discipline.

Acknowledgements

Doing research is like wandering a big city you may have heard about, but never have visited before. It has beautiful lanes, broad promenades and sky-high buildings. But there are also may narrow pathways, ending in vain, and deserted dark streets nobody seems to live. Thankfully there is always your supervisor, Erik Barendsen, standing on the corner of a street with a lantern when the sun already set. He is always willing to lead you through the dark alleys and help you find your way in that new neighbourhood you would like to explore. Also, when you get caught up in a big library with lot of interesting (but not related) things, he is there to hurry you up. The time and patience Erik devoted to me and my work are admirable.

In such a big city, you meet many new people. Jos Tolboom is one of them. Together with Erik, he gave me the opportunity to enter the city's laboratories: join the renewal commission for high school computer science education in the Netherlands. In this laboratory new and vibrant experiments contributed to this thesis. Discussion and feedback from all participants in the commission helped me to form my ideas and reflect on my thoughts. Next to this, Jos gave me the chance to present preliminary results of my expedition at the NIOC conference in Enschede.

Sometimes big cities get you caught up in new adventures, which require your full attention. Luckily, Rinus Plasmeijer put trust in me and allowed me to split time across two projects. He had patience to stand by till this thesis was fully completed. Sjaak Smetsers quickly dived into this adventure and shined his light upon my work.

Fortunately, you can always meet friends that are on the same adventure. Working together at the *department of party businesses*[1] was an experience on its own. In the weekends, *study centre Multatuli* and *monastery Westerhelling* were a welcome change in habitat. The friends you encounter at these and other places always had time for a good conversation and a nice cup of tea. Also, I am really grateful

1 As we used to call our own, trusted place in this vibrant city.

for the walks we made through the parks and the woods surrounding the city. Together you can watch the sun rising above this intriguing town of research you are not done with yet.

Tim Steenvoorden
Nijmegen, July 2015

Contents

1

Introduction

Teaching computer science at school is a hot topic. All around the world teachers, scientists and politicians are thinking about how to teach and what to teach to students at high school, middle school and primary school. In the Netherlands there is also an ongoing movement towards a new computer science curriculum at high school level.

Fortunately, the Netherlands are not alone. We can make use of already produced curricula in other countries and their approach to fundamental concepts, skills and attitudes. It is important to know how different countries approach the development of their computer science curriculum. This background can help others in forming their own curriculum on computer science. To get a good overview of all developments abroad, we need a structured instrument to compare and analyse them.

The updated Dutch curriculum must conform to a couple of important guidelines. These guidelines are imposed by the Dutch government. First of all it should be *modular* in its design. The modularity makes sure the subject can be taken by students in all four tracks provided by Dutch high schools. *Interdisciplinarity* is another a design objective. Teachers should be encouraged to work together with departments like biology, physics, arts or economics. A further goal is *sustainability*. Computer science is a young and fast moving discipline. Currently used techniques can be obsolete by ten years. Forcing students to learn possibly obsolete knowledge would be unfortunate. Therefore it is important to focus on fundamental notions and ideas.

It is these fundamental notions and ideas that make creating a computer science curriculum a difficult and defiant operation. Computer science is a new discipline, but computer science teaching methods are even younger. The questions arise what the fundamental ideas in computer science are. Furthermore, how can we design a curriculum around it? Next to providing the current state of affairs on (junior) high school computer science curricula and guidelines, we give direction in the usage of fundamental ideas in curriculum design.

Structure of this thesis

In the next chapter, Chapter 2, we study existing computer science curricula for high schools and their contents. Using a classical division in *knowledge areas*, we investigate the *content knowledge* of the documents and how it compares. After a quantitative analysis of contained concepts, we dive deeper into the documents giving an qualitative analysis of notable knowledge areas.

From Chapter 3 we take a different approach. There, we take notice of *fundamental ideas* and what criteria they have to satisfy. Applying this theory to computer science, we acquire a list of fundamental ideas.

Chapter 4 explores a way to use these fundamental ideas in the context of curriculum development. We study a contemporary framework for high school STEM subjects. After discussing the structure of the framework, we examine the applicability of it to the field of computer science and the fundamental ideas already incorporated in the framework. Subsequently, we propose to extend this framework in way it incorporates computer science as well as the original science and engineering disciplines.

In the last chapter, Chapter 5, we conclude with an overview of the contributions of this thesis.

2

Knowledge content of international science curricula

In this chapter we study the declarative knowledge of four international computer science curricula and guidelines. In comparing the different international documents, we rely on the next research question:

- *How do international curricula and guidelines differ on declarative knowledge?*

After introducing some general concepts about knowledge in §2.1, we describe a method to analyse curricula and guidelines in §2.2. We come up with an approach to code and classify concepts, based on the knowledge areas *described by the ACM/IEEE. The results of this comparison will be presented in §2.3 and further analysed in §2.4. We will end this chapter with a conclusion in §2.5.*

2.1 Theoretical background

2.1.1 *Levels of knowledge*

From a cognitivistic viewpoint, knowledge is not one uniform category. It can be divided into three levels: *declarative knowledge*, *procedural knowledge* and *metacognitive knowledge* (Dillon, 1986). In the definition below we give a short descriptions of each kind of knowledge.

Definition 2.1 *Levels of knowledge* Knowledge is split into three levels:

1. *Declarative knowledge*
   Declarative knowledge is about definitions, formulas, laws, and phenomena. This kind of knowledge is easily retrieved from knowledge sources like books and the internet. Declarative knowledge is comparatively easy to learn but hard to remember when not used actively.

II *Procedural knowledge*
This is the knowledge of *know-how*. These are method and procedures which can range from very specific (cycling), to very general (problem solving). Learning methods and procedures takes more time and is more difficult than learning declarative knowledge

III *Metacognitive knowledge*
Metacognitive knowledge is knowledge about the way we learn and think. This is sometimes referred to as *knowledge of knowledge*. Planning, evaluating and monitoring are three essential parts of metacognitive regulation (Schraw, 1998). Every time we think about our learning, we are using and developing metacognitive knowledge. Metacognition is a fairly new research discipline in educational sciences and psychology.

Declarative knowledge is sometimes called *descriptive knowledge* or *propositional knowledge*. The OECD (2013) uses the term *content knowledge*. Procedural knowledge is occasionally referred to as *imperative knowledge*.

The taxonomies of Bloom (Bloom, 1956; Krathwohl, 2002) and De Block (De Block and Heene, 1987) also make these distinctions in knowledge levels, although more elaborate. In case of Krathwohl (2002), *facts* are added to distinct them from concepts. De Block and Heene (1987) propose an even more fine grained structure, adding not only facts but also *relations* and *structures*. For our purpose the three levels presented in 2.1 and Figure 2.1 are sufficient.

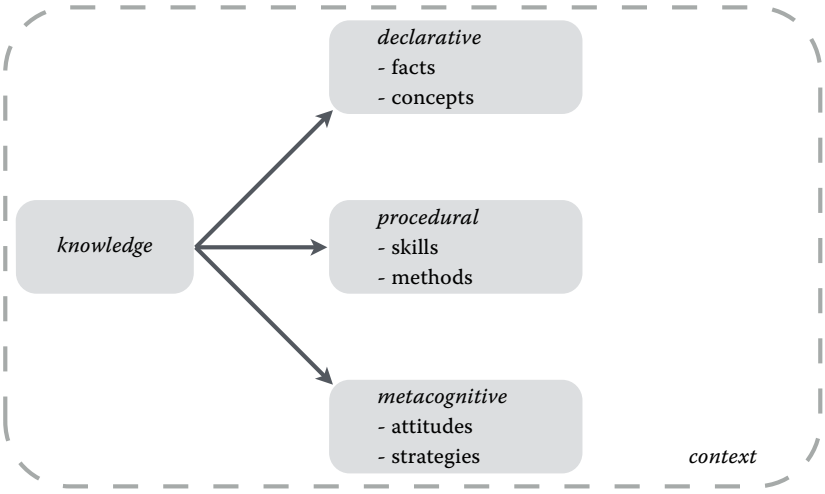


Figure 2.1 From a cognitivistic viewpoint, *knowledge* can be divided in three levels: *declarative knowledge*, *procedural knowledge* and *metacognitive knowledge*. Connected to each level are different *knowledge objects* such as *concepts*, *skills* and *attitudes*. Knowledge as a whole is learned and applied in a *context*.

In the last decade, it becomes more and more common to take the *context* of knowledge into account (Bruning and Michels, 2013). This idea emphasises that

knowledge does not stand on its own. It is always taught and used under certain conditions or in a specific environment. Boersma, Eijkelhof, van Koten, Siersma, and van Weert (2006) describe a context as '*the situation or problem statement that gives sense to students engaged in learning.*'

Each level of knowledge connects to different *knowledge objects*. As show in Figure 2.1 these are amongst others *concepts*, *skills* and *attitudes*. Throughout this text we will use these therms to designate knowledge objects from each level.

2.1.2 *Knowledge areas in computer science*

From 1968 till now the ACM and the IEEE created several curriculum guidelines in computer science, computer engineering, information systems, information technology and software engineering. The latest iteration from 2013 defines a body of knowledge for computer science, organised into a set of 18 *knowledge areas*. Each knowledge area contains *topics* and *learning outcomes* and topics are identified as either *core* or *elective*. These guidelines are mostly used by universities to design a broad undergraduate computer science curriculum. We use the knowledge areas of the ACM/IEEE as a base categorisation for the whole discipline.

2.1.3 *International curricula and guidelines*

We analysed the content of 4 curricula and guidelines[1] on computer science, all from different countries, on the declarative knowledge they incorporate. The documents we study are from:

1. France (Ministère de l'Éducation nationale, 2012)
2. CAS (CAS Working Group, 2012)
3. CSTA (CSTA Standards Task Force, 2011)
4. The Netherlands (SLO, 2007)

We include the Dutch curriculum for reference purposes. As stated in the preface, this thesis is written in the context of the renewal of the Dutch computer science high school curriculum. Therefore it is helpful to compare the present Dutch curriculum to newer curricula of surrounding countries and get a sense of the position of the Netherlands amidst them.

For this research the French and Dutch curricula had to be translated to English. The French curriculum was translated by an automatic translator,[2] but during coding activities constantly compared with the original (see also §2.2.2). The Dutch curriculum was translated by the researcher.

In the next sections we shortly describe each curriculum, including the general structure, the main goal and the intended grades. We will start each section with a quote which characterises the curriculum.

1 We will refer to these documents as *curricula* in the rest of this thesis for simplicity.
2 Google Translate: *https://translate.google.com/*.

Curriculum 1 *France*

Learn to read, write, count, reason, and program.
(Académie des Sciences, 2013, page 13)

The opinion of the curriculum authors is that programming should be on par with other standard skills learned during basic education like reading and writing. The curriculum is developed for students in the last years of their high school career (*lycée*, ages between 15 and 18 years old). It is build on four main pillars, each representing a major area of computer science. The curriculum (Ministère de l'Éducation nationale, 2012) and the accompanied document (Académie des Sciences, 2013) use a slightly different nomenclature which we reflect in Table 2.1 below.

Table 2.1 Nomenclature of the four main pillars of the French curriculum as used by the curriculum itself (left column) and by the accompanied document (right column).

Ministère de l'Éducation nationale (2012)	Académie des Sciences (2013)
Representation of information	Data
Algorithms	Algorithms
Languages and programming	Languages
Hardware architectures	Machines

The choice for these four pillars is motivated by Académie des Sciences (2013) as follows. *Algorithms* are stated as the answer on questions that start with '*How*...'. To materialise the concept of an algorithm, it has to be written in a *language*. This language has to be understandable by both humans and *machines*. Machines on their hand, execute instructions stated in a language and manipulate *information*. This covers all the activities computer scientists deal with.

The curriculum is very compactly formulated in a table listing 'knowledge', 'skills' and 'remarks'. Below we present the first two learning objectives of the section 'Representation of information' to give a feeling on the formulation.

	Knowledge	*Skills*
Binary representation	*A computer is a machine that manipulates digital values represented in binary form.*	*Handle elementary operations on three basic units: bit, byte, word.*
Boolean operations	*Introduction to the basic Boolean operations (and, or, not, exclusive-or)*	*Express simple logical operations by combination of basic operators.*

(Ministère de l'Éducation nationale, 2012)

Curriculum 2 CAS

> *Just what is computer science, viewed as a school subject?*
> (CAS Working Group, 2012, page i)

CAS divides its curriculum in 5 sections, which are:

- Algorithms
- Programs
- Data
- Computers
- Communication and the Internet

These sections mostly corresponds to the pillars used in the French curriculum: *Machines* is split across *Computers* and *Communication and the Internet*, and *Languages* corresponds to *Programs*.

The main goal is to develop a curriculum for the whole trajectory from the beginning of primary school till the end of secondary school. Therefore the curriculum differentiates learning objectives for 4 different levels, called *key stages*. As a consequence, topics are repeated. They first appear in learning objectives aimed at *key stage 1* and get deepened later on for *key stages 2, 3* and *4*.

The section on *Data*, for example, contains learning objectives about *binary switches* to store information. Students from *key stage 1*, just have to understand what they are and that a single switch encodes information:

> - *Computers use binary switches (on/off) to store information.*
> - *Binary (yes/no) answers can directly provide useful information (e.g. present or absent), and be used for decision.*
>
> (CAS Working Group, 2012, page 16)

At the end of the intended route the curriculum prescribes, students learn about the problems and limitations binary representation imposes on the encodability of information:

> - *Problems of using discrete binary representations:*
> - *Quantization: digital representations cannot represent analogue signals with complete accuracy (e.g. a grey-scale picture may have 16, or 256, or more levels of grey, but always a finite number of discrete steps)*
> - *Sampling frequency: digital representations cannot represent continuous space or time (e.g. a picture is represented using pixels, more or fewer, but never continuous)*
> - *Representing fractional numbers*
>
> (CAS Working Group, 2012, page 17)

Curriculum 3 CSTA

> *Our lives depend upon computer systems and the people who maintain them to keep us safe [...]* (CSTA Standards Task Force, 2011, page 2)

The goal of this curriculum is to teach students knowledge and skills to thrive an global information economy. Computer science spans business, science and engineering and all these problems require thorough analysis, understanding and creativity. The curriculum is built on 5 *strands*:

- *Computational thinking*
  Computing thinking is defined as '*an approach to solving problems in a way that can be implemented with a computer*' (Barr and Stephenson, 2011, page 51). It enables students to better conceptualize, analyse and solve complex problems and help them selecting and applying appropriate strategies and tools (CSTA Standards Task Force, 2011). Algorithmic thinking is regarded as a part of computational thinking.
- *Collaboration*
  Teamwork, constructive criticism, project planning and management, and team communication are all considered necessary 21st century skills. Because of this and its multiple usage in software engineering, collaboration receives its own strand in this curriculum.
- *Computing practice and programming*
  This consists of creating and organise systems and web pages, as well as exploring the use of programming in solving problems, and selecting appropriate Application Programming Interfaces (APIs).
- *Computer and communication devices*
  Because of a global impact of the internet on communication, students should be able to understand elements of modern computer and communication devices and networks and apply appropriate and accurate terminology. This contains also the organization of web pages and URLs.
- *Community, global and ethical impacts*
  This is regarded as an essential element of both learning and practice. It contains the norms of ethical use of the internet and topics about personal privacy, network security, licenses and copyright. Students, for example, should be able to make an informed and ethical choice between proprietary software and open source software.

Like the CAS curriculum, the CSTA curriculum prescribes learning objectives for different *levels*: level 1, 2 and 3. Each level corresponds to a range of grades. Learning objectives in each level are grouped in *courses*. Level 1 and level 2 each own one course, level 3 has three:

- *Level 1 (grades K–6)*
  - CS and me
- *Level 2 (grades 6–9)*
  - CS and community
- *Level 3 (grades 9–12)*
  - CS in the modern world
  - CS concepts and practices
  - Topics in CS

Courses build upon each other and only 'Topics in CS' is marked as elective.

The structuring in levels means, as with the CAS curriculum, that topics can occur multiple times. Learning objectives are, after all, repeated and deepened over

the three levels. As an example, we use (again) binary representation and usage thereof. We first encounter this in the course 'CS and community' from level 2:

> *Describe the relationship between binary and hexadecimal representations*
> (CSTA Standards Task Force, 2011, page 18)

And later on in the course 'CS concepts and practices' from level 3:

> *Discuss the interpretation of binary sequences in a variety of forms (e.g., instructions, numbers, text, sound, image).*
> (CSTA Standards Task Force, 2011, page 21)

Both learning objectives are formulated in the strand 'Computational thinking'.

Curriculum 4 *The Netherlands*

> *A country that innovates is digitally literate.* (KNAW, 2012, page 10)

The Dutch curriculum is currently being revised. The current curriculum dates from 2007, which is a modified version of the curriculum from 1995. It consists of 4 *domains*:

- A Computer science in perspective
- B Basic concepts and techniques
- C Systems and their form
- D General applicability

Domain A and domain D are about the broader, social context of computer science. The other two domains (B and C) contain computer science specific topics.

The curriculum is oriented towards grades 10 to 12 of high school. Special to the Dutch curriculum is a division in two *school types*: HAVO (higher general secondary education) and VWO (pre-university education). In the Dutch education system, students at the age of 12 go to different schools which denote the intellectual level of the students. A diploma from a specific school type gives access to a specific type of higher education. For example, HAVO gives access to vocational studies and VWO to universities. The Dutch computer science curriculum is different in that it has to take into account these two types of school.

Below we show one of the learning objectives from the Dutch computer science curriculum on databases. The first sentence is a goal for both school types (HAVO and VWO). The second, emphasised, sentence is marked as 'VWO *only*' and thus applies only to students following pre-university education.

> *The candidate can name the elements of a relational schema and describe the significance of each element, and can convert information needs into a command formulated in a query language for a relational database.* He can describe the features and aspects of database management systems, and name and use them for specific systems (VWO only). (SLO, 2007, page 3)

2.2 Method

For the content analysis of the curricula, we used a mixed method approach consisting of three phases (Cohen, Manion, and Morrison, 2011). In the first phase we created a categorisation of computer science fields, using the knowledge areas as described by ACM/IEEE (2013). Where reasonable, knowledge areas were merged into one category, mainly because of the focus on (junior) high school instead of universities. The second phase comprised coding of the curricula. Coding lead to a list of concepts mentioned in the curricula. The categories form the first phase were used to grouped all coded concepts. The third and last phase was a qualitative analysis of the curriculum texts. We used the results from the second phase as a starting point into analysing notable categories objectives in more depth.

The method's three phases will be discussed in more detail in the next sections.

2.2.1 *Categorisation of CS fields*

To compare curricula on declarative knowledge, we need a categorisation of computer science in fields. Hereto we use the curriculum guidelines of the ACM/IEEE, which provides us with *knowledge areas* of computer science. We can expect these guidelines to cover all fields of computer science. This framework is intended for higher education, especially an undergraduate degree in computer science. Not every knowledge area is as important in (junior) high school as it is at universities. Because we study high school curricula, some knowledge areas can be merged and others should be split. After taking the structure of our four studied curricula into account (as discussed in §2.1.3), we come to 13 categories. The 13 categories and the ACM/IEEE knowledge areas they incorporate are shown in Table 2.2.

The rationale behind merging or splitting knowledge areas is as follows. We split the knowledge area on 'Software Development Fundamentals'. This area groups four sub areas: 'Algorithms and Design', 'Development Methods', 'Fundamental Data Structures' and 'Fundamental Programming Concepts'. Each of them contain fundamental concepts of computer science. They are grouped together by ACM/IEEE to stress their importance as a foundation for all other knowledge areas. For (junior) high school it is this knowledge area which will get the most attention. Therefore we assign each of the sub areas to their own category (*algorithms*, *engineering*, *data* and *programming*) which are extended by other ACM/IEEE knowledge areas. For example: 'Algorithms and Complexity' and 'Parallel and Distributed Computing' are merged into *algorithms*, and 'Programming Languages' and its extensions 'Platform Based Development' into *programming*. Both group basic and advanced topics of the same area together into one category. Concepts about the software development process, its tools and its methods (such as collaboration) are classified in *engineering*. All of this is summarized in below table.

When comparing these four categories to the structure of our studied curricula, we encounter they cover most of the sections used by France, CAS and the CSTA.[3]

Table 2.2 Categories used to group computer science declarative knowledge based on the knowledge areas defined by ACM/IEEE (2013). Knowledge areas are merged due to our accent on (junior) high school instead of universities, which is the focus of the ACM/IEEE *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. The knowledge area 'Software Development Fundamentals' (SDF) is divided amongst four other categories because of its founding characteristics towards the remaining knowledge areas.

Category	Contained knowledge areas from ACM/IEEE (2013)
Algorithms	Algorithms and Complexity (AL)
	Parallel and Distributed Computing (PD)
	Algorithms and Design (SDF/AL)
	Remark: concepts about data structures are covered by *Data*
Architecture	Architecture and Organization (AR)
	Operating Systems (OS)
	System Fundamentals (SF)
Data	Information Management (IM)
	Fundamental Data Structures (SDF/IM)
Engineering	Software Engineering (SE)
	Development Methods (SDF/SE)
	Remarks: contains also concepts on collaboration; concepts without an engineering component are covered by *Programming*
Graphics	Graphics and Visualisation (GV)
Intelligence	Intelligent Systems (IS)
Mathematics	Discrete Structures (DS)
Modelling	Computational Science (CN)
Networking	Networking and Communication (NC)
Programming	Programming Languages (PL)
	Platform Based Development (PBD)
	Fundamental Programming Concepts (SDF/PL)
Security	Information Assurance and Security (IAS)
	Remark: concepts about privacy are covered by *Society*
Society	Social Issues and Professional Practice (SP)
Usability	Human-Computer Interaction (HCI)

We miss, however, categories on *machines*[4] and *networks*. These are covered by the knowledge areas 'Architecture and Organization', 'Operating Systems', 'System Fundamentals' and 'Networking and Communication'. To make a clear distinction between the architecture of a device and the communication between devices, we group the first three into a category named *architecture* and put the last one into

3 We refer to §2.1.3 for more information on the structure of the curricula.

4 Referred to by some curricula as *devices* or *computers*.

its own category *networking*. All knowledge areas in *architecture* describe lower level organisational strategies and system architectures and the one in *networking* describes the strategies to connect these systems together. Other knowledge areas are to distinct, even on a high school level, and are used as-is.

2.2.2 *Coding concepts in curricula*

Concepts were coded to detect the declarative knowledge contained in each curriculum. During the coding process, codes were adapted to create a common set of concepts, applicable to every curriculum (axial coding, Strauss, Corbin, and Others (1990)). For example, the CAS curriculum formulates a learning objective about ways '*information can be represented*' (CAS Working Group, 2012, page 16). The CSTA phrases this as '*representation [...] of digital information*' (CSTA Standards Task Force, 2011, page 18) and France as '*digitalisation*' (Ministère de l'Éducation nationale, 2012). All three formulations denote the same concept, and are coded with the label *information representation*. Such co occurrences were merged or linked to reflect their relationship. Coding was done using quantitative data analysis software.[5] Only the parts of the curricula containing learning goals were taken into account, the accompanying text was not considered.

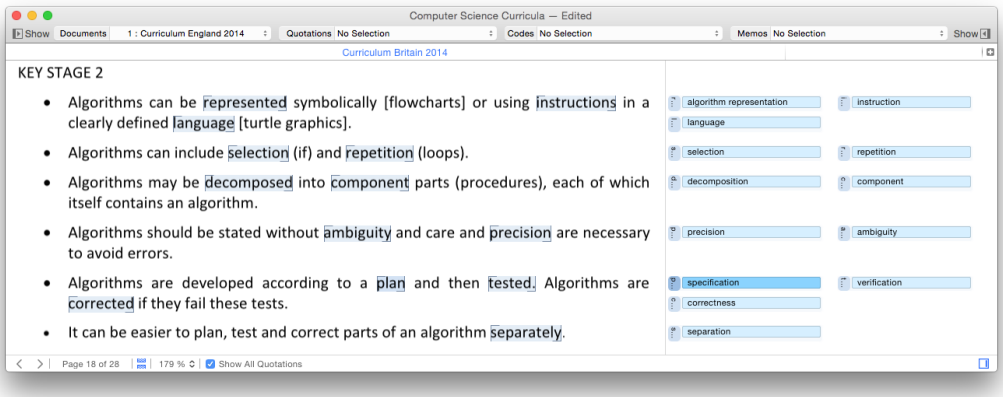


Figure 2.2 Excerpt from the CAS curriculum showing quotations and associated labels. The objectives come from the section titled '*Algorithms*' and are directed towards key stage 2 students (children between 7 and 11 years old).

To demonstrate how we distilled concepts from the learning goals, we study an example. Figure 2.2 shows an excerpt from the CAS curriculum. The learning objectives are part of the section about algorithms and directed toward key stage 2 students (children with an age between 7 and 11 years). Each learning objective

5 Atlas.ti for Mac, version 1.0.24: *http://atlasti.com/product/mac-os-edition/*.

contains the concept *algorithm*, but it is not the main goal intended by each objective and therefore not coded as such. The first objective is about *representing algorithms*, using *instructions* and *languages*. The associated quotes are highlighted in the text and linked to a corresponding label as shown in the right margin. We apply the same argumentation to all next learning objectives. The last two however need special attention There the words 'plan' and 'tested' refer to the concepts *specification* and *verification* as implied by their context. The last objective mentions the words 'plan' and 'test' again, but it addresses the concept of *separation of concern*.

Found concepts were grouped in the categories defined in Table 2.2 by using the topics and learning outcomes described by the ACM/IEEE. The concept *sorting algorithm*, for example, occurs at two places in the ACM/IEEE curriculum: at 'Fundamental Data Structures and Algorithms (AL)' and 'Processing (CN)'. The first topic area is about the implementation and complexity of algorithms. In the second topic area, sorting is mentioned as a example of data processing. Because the meaning of the concept *sorting algorithm*, as imposed by the high school curricula, is that of implementing and analysing it, this concept fits in 'Fundamental Data Structures and Algorithms (AL)' and is therefore filled in the category *algorithms* (see also Table 2.2). This categorisation of codes resulted in a list of concepts with linked quotations for each category defined in the previous section and for each studied curriculum.

2.2.3 *Qualitative analysis of curricula*

We regard the number of concept occurrences in the categories as an indicator for the (relative) importance of those categories in a curriculum. For example, in case the concept occurrences in 'algorithms' outweigh those in 'engineering' by far, the curriculum is likely to have a focus towards fundamental computer science. Of course concepts alone do not determine the content of a curriculum. Most curricula also mention skills and attitudes a student should be aware of. Another remark is the implicit incorporation of concepts. A curriculum can, for example, desire students to program in an imperative programming language. To do so, students have to know about the concepts *sequence*, *repetition* and *selection*. However, the curriculum does not need to mention these concepts, they are already implied by the objective to program in an imperative way. That means a curriculum can have a lower count of concepts in a group, while actually it is as thorough as others on the subject. To obviate these issues, we did not only count concept occurrences (phase two), but also analyse the curriculum content in more depth (phase three), using the frequencies of the respective categories as pointers to interesting text segments. This qualitative analysis of the curriculum texts complements the quantitative analysis from the first two phases.

2.3 Results

We present our results in three forms. Firstly, we show the distribution of quotations across all categories for every curriculum in Figure 2.3. Secondly, in Box 2.1

we list for each curriculum all categories sorted from highest to lowest number of quotations. Thirdly, all found concepts per category can be found in Chapter B.

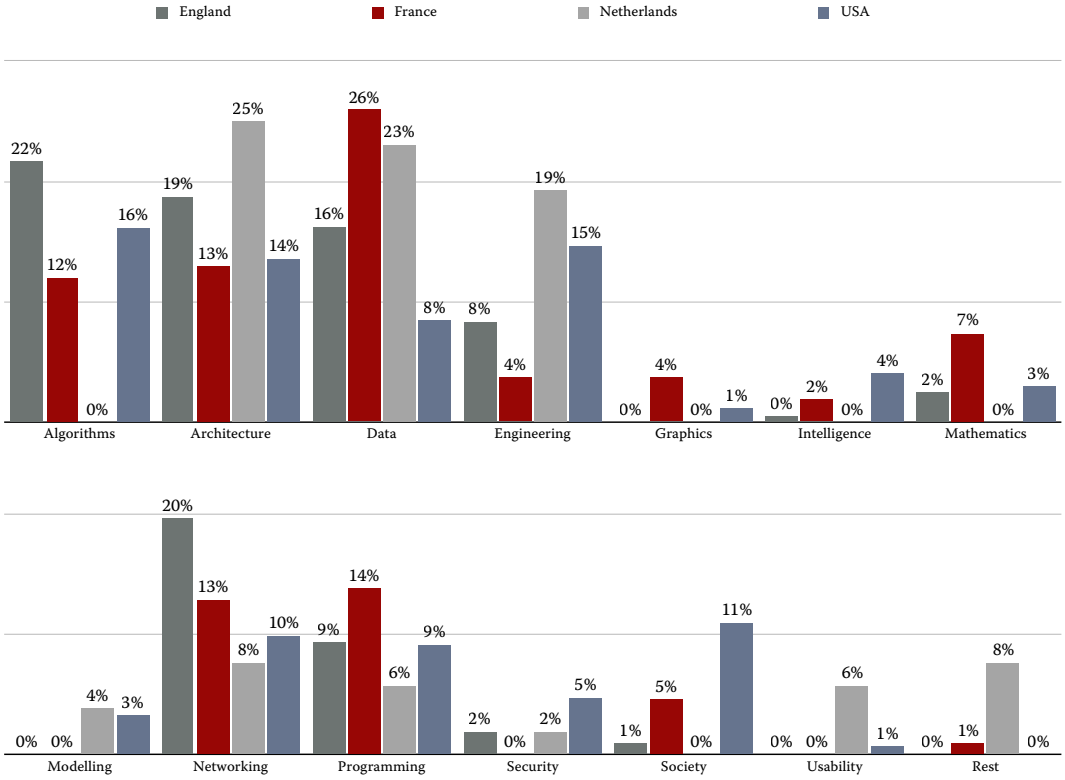


Figure 2.3 Percentage of quotations that fall into a category for all four curricula.

2.4 Analysis

Figure 2.3 gives us a quick overview how the four studied countries compare on the thirteen different knowledge categories, including a rest category. We see *algorithms*, *architecture*, *data* and *networking* cover the biggest parts of these high school curricula. These four categories all get a place in the top four of CAS and France, as we can see in Box 2.1. Although *algorithms* have the highest overall score, the Dutch curriculum does not mention anything from this category. Notable is the focus on *data* in the French curriculum and the big gap between the number of quotations in this category, and the next: *algorithms*. Another interesting point is the diverse scores in the *engineering* category: the French curriculum keeps behind the other three curricula. Exceptional is the high score on society in the CSTA curriculum when comparing this category to the other countries and the high score of the Dutch curriculum in the rest category. Last noteworthiness is the total number of quotations coded in each curriculum, as shown at the bottom of Box 2.1. The Dutch curriculum contains almost a fifth of the coded quotations when compared to the CSTA curriculum. In the next sections we will analyse all these points in more depth.

Box 2.1 Rankings of categories in each curriculum sorted from most to least occurring quotations. Between parenthesis are the number of quotations appearing in the category. The total number of quotations is given at the end of each list.

France

1 Data (28)
2 Programming (15)
3 Architecture (14)
Networking (14)
4 Algorithms (13)
5 Mathematics (8)
6 Society (5)
7 Engineering (4)
Graphics (4)
8 Intelligence (2)
9 Rest (1)
10 Modelling (0)
Security (0)
Usability (0)

(Total: 108)

The Netherlands

1 Architecture (13)
2 Data (12)
3 Engineering (10)
4 Networking (4)
Rest (4)
5 Programming (3)
Usability (3)
6 Modelling (2)
7 Security (1)
8 Algorithms (0)
Graphics (0)
Intelligence (0)
Mathematics (0)
Society (0)

(Total: 52)

CAS

1 Algorithms (44)
2 Networking (40)
3 Architecture (38)
4 Data (33)
5 Programming (19)
6 Engineering (17)
7 Mathematics (5)
8 Security (4)
9 Society (2)
10 Intelligence (1)
11 Graphics (0)
Modelling (0)
Usability (0)
Rest (0)

(Total: 203)

CSTA

1 Algorithms (44)
2 Engineering (40)
3 Architecture (37)
4 Society (30)
5 Networking (27)
6 Programming (25)
7 Data (23)
8 Security (13)
9 Intelligence (11)
10 Modelling (9)
11 Mathematics (8)
12 Graphics (3)
13 Usability (2)
14 Rest (0)

(Total: 272)

2.4.1 *Data*

Relative to other countries, France spends the biggest part of its curriculum to teach about data (26%). The category has the highest position in the ranking from Box 2.1. Remarkable is the distance to *programming* (15%, number two on the list), which is 12% (13 quotations). This can be explained by the structure of the curriculum. Almost all (19 of 28) of the coded quotations in this category come from the section 'Representation of information'. This is the biggest section in the curriculum, containing 8 of the in total 21 learning objectives (more than a third). Of the remaining quotations, 7 come from the section on 'Languages and programming'. Here the curriculum explicitly states which data types students should know.

Data types	*Integer; floating point; boolean; natural number; array; string.*	*Choosing a data type based on a problem to solve.*

(Ministère de l'Éducation nationale, 2012)

Retrieval and *storage of data* are filled under the section on 'Computer architecture'.

There are more differences in content when comparing the French section on data to, for example, the CAS one. The French curriculum includes objectives about *document formats* and *directory structure*, only parred by the CSTA curriculum. Document formats are discussed in general, but also specific instances for images and audio.

Formats	*Digital data is arranged to facilitate storage and processing. The structuring of digital data respects either de facto standards or norms.*	*Identify some document formats, images and sound data. Choose an appropriate format compared to a use or need, quality or limitations.*

(Ministère de l'Éducation nationale, 2012)

The curriculum also mentions explicitly students have to learn about the representation of characters, text, numbers, floating points and images.

Digitalisation	*The computer handles only numeric values. A digitalisation step of physical world objects is essential.*	*Encode a number, a character through a standard code, a text in the form of a list of numeric values. Encode an image or sound as an array of numeric values. [...]*

(Ministère de l'Éducation nationale, 2012)

The CAS and CSTA curricula only mention *information representation* in general and *binary representation*.

> 6 *Analyze the representation and trade-offs among various forms of digital information.* (CSTA Standards Task Force, 2011, page 18 (level 3))

The Dutch is lacking both.

Exceptionally the CAS curriculum contains a learning objective on *representation sharing*: the fact that one sequence of bits can be interpreted in multiple ways and yield a number, string or audio fragment dependent on the intended usage.

- *Many different things may share the same representation, or "the meaning of a bit pattern is in the eye of the beholder" [e.g. the same bits could be interpreted as a BMP file or a spreadsheet file; an 8-bit value could be interpreted as a character or as a number].*
  (CAS Working Group, 2012, page 16 (key stage 3))

The curriculum also includes concepts on *data errors* and *sampling frequency*.

The high score on data by the Dutch curriculum can be attributed to the learning objectives on *information systems*, *databases*, *relational schemas* and *query languages*.

> 15 *The candidate can name the elements of a relational schema and describe the significance of each element, and can convert information needs into a command formulated in a query language for a relational database. He can describe the features and aspects of database management systems, and name and use them for specific systems (VWO only).*
> (SLO, 2007, page 3)

All these concepts are absent from the other three curricula.

The CSTA is the only curriculum which mentions *big data*, although not explicitly.

> 4 *Compare techniques for analyzing massive data collections.*
> (CSTA Standards Task Force, 2011, page 18 (level 3))

2.4.2 *Algorithms*

The CAS curriculum is special in that it explicitly states *sequence*, *selection* and *repetition* as concepts in its curriculum text.

- *The idea of a program as a sequence of statements written in a programming language [Scratch].*
- *One or more mechanisms for selecting which statement sequence will be executed, based upon the value of some data item.*
- *One or more mechanisms for repeating the execution of a sequence of statements, and using the value of some data item to control the number*

of times the sequence is repeated.
(CAS Working Group, 2012, page 14 (key stage 2))

The CSTA curriculum goes even further and, instead of *repetition* in general, states *iteration* and *recursion*.

3 *Explain how sequence, selection, iteration, and recursion are building blocks of algorithms.*
(CSTA Standards Task Force, 2011, page 18 (level 3))

It is the only curriculum which includes recursion in its learning objectives. This is in contrast to France and the Netherlands, which do not explicitly name these. Likewise, CAS and the CSTA make the underlying notion of an *instruction* explicit, where France and the Netherlands do not.

- *A computer program is a sequence of instructions written to perform a specified task with a computer.*
(CAS Working Group, 2012, page 14 (key stage 2))

The notion of an algorithm is a key concept in the CAS curriculum. The definition of an algorithm returns at every key stage, each time adding more details appropriate for the intellectual level of the students.[6]

- *Algorithms are sets of instructions for achieving goals, made up of predefined steps [the 'how to' part of a recipe for a cake].*
- *Algorithms can be represented symbolically [flowcharts] or using instructions in a clearly defined language [turtle graphics].*
- *An algorithm is a sequence of precise steps to solve a given problem.*
- *The choice of an algorithm should be influenced by the data structure and data values that need to be manipulated.*
(CAS Working Group, 2012, page 13–14 (key stages 1–4))

The gradual nature of the CAS curriculum could possibly lead to multiple similarly coded quotations. This imposes restrictions on our analysis. To preclude this, we list all different concepts on algorithms in both the French and the CAS curricula in Box 2.2.

We observe the CAS curriculum contains almost three times as much different concepts on algorithms than the France curriculum. We can conclude that CAS provides more in depth concepts in the algorithm category. Most notable is the inclusion of *concurrency* and *parallelism* in the CAS curriculum.

Although *searching* and *sorting* are included by France, CAS and the CSTA, the French curriculum is the only one of the three making some algorithms explicit. It contains *merge sort*, *breadth first search* and *depth first search*.

6 The fragments between brackets are from the original and are intended as example.

Box 2.2 Concepts on algorithms appearing in the French and CAS curriculum.

France	CAS	
• algorithm	• algorithm representation	• instruction set
• breadth first search	• ambiguity	• live-lock
• data processing	• component	• output
• depth first search	• concurrency	• performance
• finite state machine	• data processing	• precision
• instruction set	• deadlock	• problem solving
• merge sort	• decomposition	• redundancy
• search algorithm	• decision	• repetition
• sort algorithm	• input	• search algorithm
	• instruction	• selection
	• instruction sequence	• sort algorithm
		• step
		• task

Advanced algorithms	*Merge sort; search for a path in a graph by a depth first search (DFS); finding a shortest path through a wide path (BFS).*	*Understand and explain (orally or in writing) an algorithm. Questioning the effectiveness of an algorithm*

(Ministère de l'Éducation nationale, 2012)

Exceptionally, the French curriculum is the only curriculum including *finite state machines*.

> *...describe a single event system with a finite state machine.*
> (Ministère de l'Éducation nationale, 2012)

The Dutch curriculum does not contain any concepts from the algorithm category. Some of the concepts in this category are implicit in the learning objective on software development.

> 7 *The candidate mastered simple data types, program structures and programming techniques.* (SLO, 2007, page 2)

However, by not mentioning algorithmic capabilities explicitly, the teaching of these subjects are not mandatory and are not tested during examinations.

2.4.3 *Engineering*

The French curriculum has a notably low score in category engineering (4%) when comparing to the other countries. It does contain pointers to *testing* and *verification*.

Fixing a program	*Test; instrumentation; error situations or bugs.*	*Testing a developed program.* Optional*: using a development tool.*

(Ministère de l'Éducation nationale, 2012)

Concepts on testing and verification can be found in all other curricula, except for the Dutch curriculum. Its high score in this category is due to the inclusion of project management and related concepts like *specification*, *requirement*, *client* and *prototype*.

17 *The candidate can asses progress of a simple system development process, test prototypes, make sure the final product meets the specifications of the client and evaluate whether the system meets the requirements.*
(SLO, 2007, page 3)

In the section on programming in the CAS curriculum, we find various learning objectives stating concepts about engineering spread amongst multiple key stages.

- *Programs are developed according to a plan and then tested. Programs are corrected if they fail these tests. The behaviour of a program should be planned.*
- *Documenting programs to explain how they work.*
- *Programs are developed to meet a specification, and are corrected if they do not meet the specification. Documenting programs helps explain how they work.* (CAS Working Group, 2012, page 15 (key stages 2–4))

Although the curriculum of the CSTA does not explicitly state concepts like *specification* and *requirement*, it does mention the *software development process* and *software life cycle* and the creation of problem statements in general.

2 *Describe a software development process used to solve software problems (e.g., design, coding, testing, verification).*
(CSTA Standards Task Force, 2011, page 18 (level 3))

Furthermore, the CSTA curriculum has a very strong focus on collaboration during software development. This is not surprising when taking the structure of the curriculum into account. One of the five strands the curriculum is built on is 'Collaboration' and a big part of the curriculum is dedicated to it. Accompanied concepts next to *teamwork*, *collaboration* are *peers*, *experts*, *pair programming*, *project teams*, *feedback*, *communication*, *feedback* and *socialization*.

3 *Identify ways that teamwork and collaboration can support problem solving and innovation*
(CSTA Standards Task Force, 2011, page 14 (level 1))

3 *Collaborate with peers, experts, and others using collaborative practices such as pair programming, working in project teams, and participating in group active learning activities.*

4 *Exhibit dispositions necessary for collaboration: providing useful feedback, integrating feedback, understanding and accepting multiple perspectives, socialization.*
(CSTA Standards Task Force, 2011, page 16 (level 2))

The curriculum also mentions multiple productivity tools, development tools and collaboration tools explicitly.

1 *Use productivity technology tools (e.g., word processing, spreadsheet, presentation software) for individual and collaborative writing, communication, and publishing activities.*
(CSTA Standards Task Force, 2011, page 13 (level 1))

2 *Use collaborative tools to communicate with project team members (e.g., discussion threads, wikis, blogs, version control, etc.).*
(CSTA Standards Task Force, 2011, page 19 (level 3))

The inclusion of collaboration and tools make the CSTA curriculum stand apart from the other three. Although CAS the Netherlands and the CSTA all spend a reasonable part of their curriculum to engineering, only the CSTA consider collaboration as a main component of computer science.

2.4.4 *Society*

A major difference between the CSTA and the other three countries is the focus on computer science and society (11%). As with collaboration, this is also one of the five strands of the curriculum. Although CAS and France mention privacy (EN) or *personal information* and *ownership* (FR), they leave it to that.

Persistence of information	*Data, including personal, may be stored for long periods without control by the persons concerned.*	*Awareness of the persistence of information on digital networks. Understand the general principles to behave responsibly in relation to the rights of individuals in digital platforms.*

(Ministère de l'Éducation nationale, 2012)

The CSTA go further with the inclusion of 27 other concepts ranging from *career perspectives*, via different *software licenses* to *software piracy* and *legal behaviour*.

1 *Exhibit legal and ethical behaviors when using information and technology and discuss the consequences of misuse.*
(CSTA Standards Task Force, 2011, page 17 (level 2))

2.4.5 *Rest*

Exceptional in Figure 2.3 is that the Dutch curriculum has a high score in the rest category (8%). It has to do with the fact that this curriculum contains subjects on management and organisation structures which are not mentioned in the ACM/IEEE body of knowledge. The Dutch curriculum explicitly states students should know about project management and business structures as we can see in the next learning objective from domain B, 'Basic concepts and techniques'.

> 8 *The candidate knows the overall organizational structure of companies. He knows the characteristics of a project and can indicate why, during major changes in a information system of a company, one often chooses to use a project.* (SLO, 2007, page 2)

No other curriculum mentions these kind of objectives.

The one concept filled in the rest category from the French curriculum is due to a reference to *standards*. In this context, standards indicate technical global guidelines by, for example, the IEEE.

2.4.6 *Number of quotations*

The total number of coded quotations is give at the end of each list in Box 2.1. The reason that France and the Netherlands have less coded quotations, is because the learning goals are very compactly formulated and concepts often are mentioned just once (see also §2.1.3). CAS and the CSTA put together their curriculum in a more constructive way, first formulating learning goals for lower grades and after that for higher grades. As an example, we show the learning objective about information representation from the Dutch curriculum.

> 5 *The candidate can describe and apply common digital encodings of data.* (SLO, 2007, page 2)

Where the CAS French and CSTA curricula fill multiple sections on the matters of data, information and representation, the Dutch curriculum states one line and thus containing much less concepts.

2.5 Conclusion

For France, we encountered mostly concepts about data, which is no surprise when taking the structure of the curriculum into account. Furthermore it is notable that the whole curriculum is fairly theoretically oriented, focussing on concepts of fundamental computer science leaving engineering practices but especially social aspects behind.

We already noted that the overall structure of the CAS curriculum resembles that of the French. Taking this into account, the CAS curriculum seems to focus on the theoretical foundations of computer science, as does the French. However

it does take software engineering and a small amount of security and society into account. Another main difference is the focus on algorithmic thinking instead of data and information representation.

The CSTA presents us with a practically oriented curriculum. Software engineering and social and ethical topics cover together a quarter of the curriculum. This focus is also reflected in the structure of the curriculum. That does not mean this curriculum comprises on fundamental computer science. Computational thinking is broadly represented.

The Dutch curriculum is exceptional in two ways. Firstly, it does not contain any concepts in the category on algorithms. Secondly, it includes concepts about databases, project management and business which is not included in any other studied curriculum.

In addition to a difference in declarative knowledge, our research also produced a list of computer science concepts, grouped in 13 categories, which should be taught to (junior) high school students according to 4 international curricula. This list can be used in other studies like Barendsen, et al. (2015) to analyse key concepts in K-12 education and their assessment.

3

Finding fundamental ideas in computer science

In this chapter we introduce fundamental ideas*, ideas that are at the basis of a discipline (Bruner, 1960). We will take a look at why fundamental ideas are important for learning and how they can help building a curriculum in §3.1. Fundamental ideas are a way to explore a discipline, but we need tools to recognize and categorize them. These tools are introduced in §3.2 and utilised in §3.3 where we present some fundamental ideas of computer science.*

We address the following questions in this chapter:

- *What are fundamental ideas and how do we recognize them?*
- *What are the benefits of fundamental ideas in designing a (computer science) curriculum?*
- *What are fundamental ideas of computer science?*

3.1 Understanding fundamental ideas

As discussed in Chapter 2, a concept is part of *declarative knowledge*. ISO 1087-1 (2000) defines it as '*a unit of knowledge created by a unique combination of characteristics*'. Although Schwill (2004) tries to differentiate *concepts* from *ideas*, it is a very difficult task. The notion of an idea dates back to Plato and is discussed by many philosophers. In Platonic thought it is defined as '*an eternally existing pattern of which individual things in any class are imperfect copies*' (Oxford University, 2010). Ideas are thus much broader than concepts or procedures.

In 1960 Bruner published his well known book on developmental psychology entitled *The Process of Education*. Although his work mainly investigates the development of young children, it had a big impact on educational studies and curriculum studies as well. He formulated the principle that lessons should be orientated towards what he called *fundamental ideas*. Also, he argued curricula should be build around these fundamental ideas. It will help to make a curriculum more sustainable and complete.

3.1.1 *Applicability between and inside fields*

There are several advantages Bruner attributes to fundamental ideas. They have '*wide as well as powerful applicability*' (Bruner, 1960, page 18). Thus fundamental ideas have to be part of multiple fields.[1] Students can easily *extend* their knowledge to *other* fields by structuring new concepts around already known fundamental ideas. When students understand the fundamental principles, concepts and ideas they will be able to create a better and stronger cognitive structure about the whole discipline. Subject in other fields will therefore be more comprehensible.

Bruner states that

> *...intellectual activity anywhere is the same, whether at the frontier of knowledge or in a third-grade classroom.* (Bruner, 1960, page 14)

This implies there must be some kind of general principles connecting earlier learned knowledge with subjects learned at an later age. Thus fundamental ideas have to extend on all intellectual levels. Students can continually *deepen* their knowledge *inside* a particular field. When students are introduced to new concepts and principles inside a particular field, these are easier to learn. As with new knowledge in other fields, new principles inside a field are easily enriched in their existing cognitive structure.

3.1.2 *Transfer to new situations*

Next to the connections among and inside fields, fundamental ideas improve the ability to apply knowledge under new or different conditions. Knowledge learned during school, college and university can only be used later on by transferring this knowledge to new situations. Transfer comes in two forms: *specific transfer* and *non-specific transfer* (Bruner, 1960). Both are schematically represented in Figure 3.1.

Specific transfer means students can apply learned knowledge to situations which resemble already known ones. Maybe the student has to change or extend his solution schema to be applicable in the new situation, but only slightly. Schemas usable for specific transfer, are usually applicable in a narrow, well defined field. Specific transfer is mainly applied by continuing and vocational education.

Knowledge that is not taught in a form to applied immediately, is usable for non-specific transfer. It is taught in a way to be applied on the long term. Non-specific transfer is based on knowledge of fundamental notions and principles and ways of thinking. Students learn and develop attitudes to, for example, problem solving, research, heuristics, observations and so on. This kind of transfer requires a *meta cloud* to store this knowledge. Where specific transfer relates directly to a new situation, non-specific transfer goes through this meta cloud. Non-specific transfer is mainly taught by universities.

1 We use the term *field* to characterize a part of a discipline.

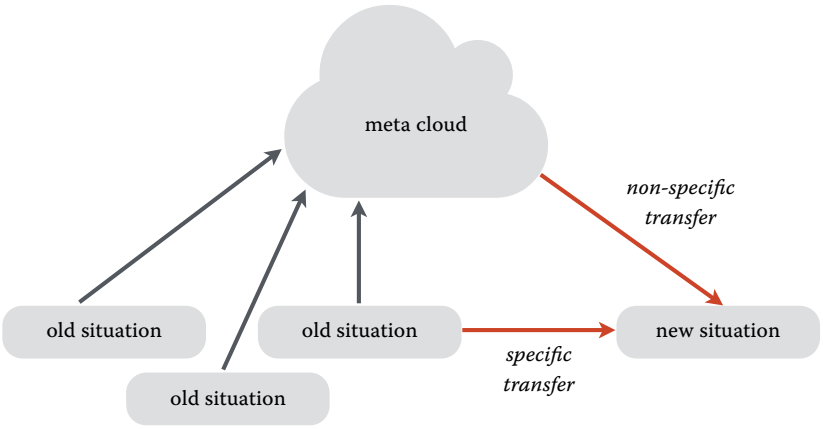


Figure 3.1 Transfer directly from an old situation to a new situation is called *specific*. Fundamental notions, attitudes and ways of thinking are not directly applicable in new situations. Instead they are stored in a *meta cloud* (grey arrows). We call transfer *non-specific* when knowledge from this meta cloud is applied to a new situation. (Bruner, 1960; Schwill, 1994)

Because fundamental ideas are so general, they allow many problems to be treated as special cases. Fundamental ideas condense information into a meta cloud which will be kept in mind for a longer time. It is also more easy to construct details out of this big structure.

3.1.3 *In context of knowledge*

In §2.1 we discussed the terms *concepts*, *skills* and *attitudes*. Each of them are tied to a particular level of knowledge, respectively *declarative knowledge*, *procedural knowledge* and *metacognitive knowledge*. In this chapter we encountered a new term, that of an *idea*. In contrast to concepts, skills and attitudes, an idea is generally not tied to one specific level of knowledge: it cuts across all three levels as depicted in Figure 3.2 This supports the view of Bruner, that ideas can be used to easily extend understanding of a subject in every possible way. Ideas can be used to learn new concepts, but it can even so be used to acquire a new skill or acquaint a new attitude.

Example 3.1 *Resources* Take for example the idea of a *resource*. Several questions about resources come to mind:

- What is a resource?
- Where to find a resource?
- How to process it?
- When is a resource trustworthy?

The first question is about declarative knowledge and, using our definition from page 3, about the *concept* ‘resource’. The next two encourage the development of new *skills*, namely how to use resources. The last question is not

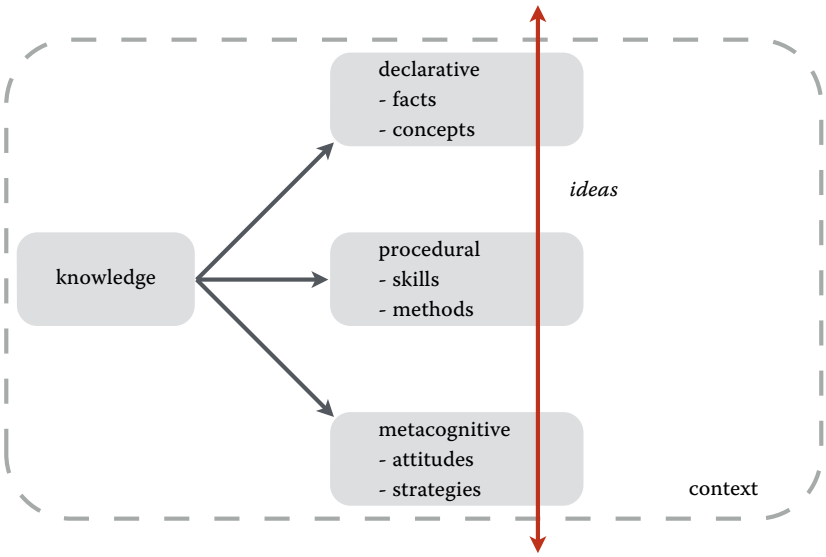


Figure 3.2 Ideas span all three levels of knowledge: declarative, procedural and metacognitive.

just about a skill ('How to judge when a resource is trustworthy?'), but also about an *attitude* towards resources ('When do I think a resource is trustworthy and why?'). It is about *evaluating* knowledge from others and belongs in the level of metacognitive knowledge.
We can conclude the idea *resource* can be used to teach knowledge at all three levels defined in §2.1. Not just concepts, skills and attitudes come into play as well.

3.2 Criteria for fundamental ideas

Bruner can give us a feeling about fundamental ideas. He shows intuitively what they are, how they can be used, what their purpose is, how they can improve learning and curriculum design and so on. He does not, however, give us a formal definition of a fundamental idea, or some guides to recognize them. In fact he used the terms *fundamental idea*, *basic idea* and *general principle* interchangeable. Schwill (1994) extends the intuition of Bruner by giving four criteria a fundamental idea should satisfy. These are the criteria of *width*, *depth*, *sense* and *time*.[2] Each of these criteria is discussed in the following sections. Along each description we show an example of a fundamental idea satisfying the particular criterion.

3.2.1 *Criterion of width*

First of all, a fundamental idea has to be applicable in a wide area of the discipline: it has to be relevant to multiple fields and appear across the entire discipline. In

2 Schwill (1994) calls the first two criteria the *horizontal criterion* and the *vertical criterion*. We like a more consistent naming scheme.

other words, if a concept is only used in a very narrow domain, it cannot be a fundamental idea. Schwill visualises this by a horizontal axis, penetrating a great number of application fields as shown in Figure 3.3.

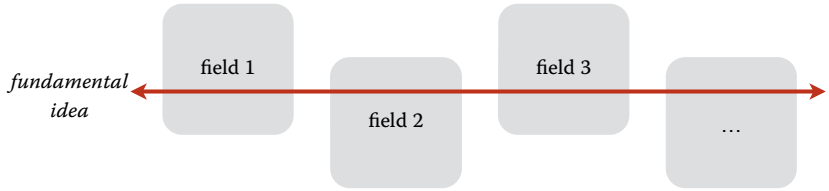


Figure 3.3 A fundamental idea has to appear across the entire discipline. Thus it should appear in multiple fields of the domain.

Example 3.2 *Process* ACM (2012) acknowledges twelve main fields of computing in their *Computing Classification System*. These are:

- Hardware
- Computer systems organisation
- Networks
- Software and its engineering
- Theory of computation
- Mathematics of computing
- Information systems
- Security and privacy
- Human-centred computing
- Computing methodologies
- Applied computing
- Social and professional topics

The term *process* is applicable in every field stated above. In the context of *hardware* we can discuss signal processing. When using *networks*, protocols and other communication processes play an important role. The scheduling of processes by an operating system is a part of *software and its engineering*. *Mathematics of computing* covers stochastic processes. For each field we can give an examples of the usage and importance of this idea. Hence we can say *process* satisfies the criterion of width in computer science and is a candidate for a fundamental idea.

3.2.2 *Criterion of depth*

Next to applicability in multiple fields, a fundamental idea has to be of great importance inside a particular field. That is, it should play a central role in that field. A fundamental idea always relevant at a very basic level of understanding, as well as on a advanced level. Not only should a fundamental idea be explainable to children in easy terms, it should also be used by professionals and researchers on a much higher intellectual level. Bruner hypothesizes that

> *...any subject can be taught effectively in some intellectually honest form to any child at any stage of development.* (Bruner, 1960, page 33)

A fundamental idea can thus be taught at every school level: at the level of primary school as well as at the level of universities. Only the amount of detail and formalisation differ. Formulated the other way around: topics that cannot be taught to students in primary school cannot be fundamental ideas (Fischer, 1984).

This idea is visualised by Schwill by a field, divided in multiple intellectual levels, from low to high. A vertical axis representing a fundamental idea pierces the field, as shown in Figure 3.4.

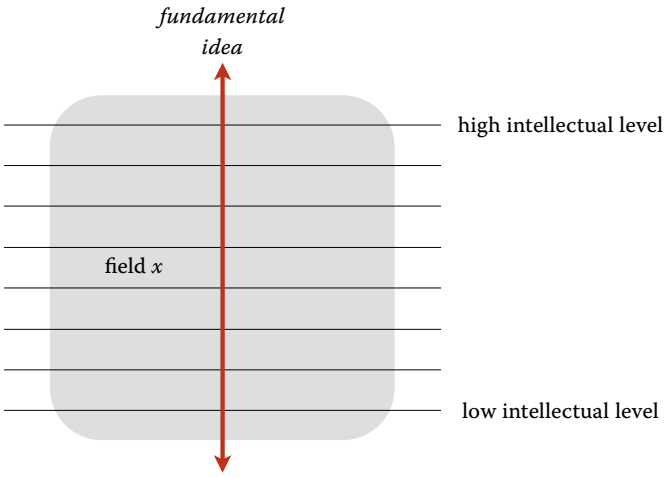


Figure 3.4 A fundamental idea should be apparent at every level of understanding inside a particular field. The vertical axis, representing the fundamental idea, pierces all intellectual levels of the field, from low to high.

The criterion of depth is, in particular, relevant to lessons and curricula (Bruner, 1960; Schwill, 1994). When an idea satisfies the criterion of depth, it can serve as a guideline on every level of the educational process. This leads to the concept of the *spiral curriculum*, in which topics are revisited several times throughout a curriculum or a course (Harden, 1999).

Example 3.3 *Worst/best/amortized case analysis* Complexity is a main part of computer science. Let us examine if it satisfies the criterion of depth and can qualify as candidate for a fundamental idea. *Worst case analysis* and *best case analysis* can already start in primary school using questions like: 'How long does it take to get to school in the worst case? For example if the bus is late, if all traffic lights are red, if the roads are icy? And how long does it take in the best case? If there is no traffic jam, if you put all your energy in cycling, and so on?' During middle school and high school students can develop a more elaborate approach to worst case analysis. They can relate the number of steps needed to find a name in a phone book and compare different approaches like linear search and binary search.[3] In other words, they can correlate the runtime of an algorithm with the length of the input. At the end of

3 Example taken from the massive open on-line course CS50 (*http://cs50.tv/*) by David J. Malan.

high school or in university, teachers can show the formal definition of worst, average and *amortized case*, order etcetera. These explenations make worst, best and amortized case analysis meet the requirements of the criterion of depth and give us new candidates for fundamental ideas.

3.2.3 *Criterion of sense*

The next criterion is about '*embodiment in everyday life*' (Schreiber, 1983). According to Schreiber, a fundamental idea is characterized by its usage in a context which is practical and unscientific.

We can visualize this everyday thinking argument by a Venn diagram of several contexts. We have to be aware of the fact that these context are practical, for example home, holiday, family and so on. A fundamental idea should be usable in all of these contexts. So the place where all the contexts overlap,[4] is the where the fundamental idea prevails. This is show in Figure 3.5.

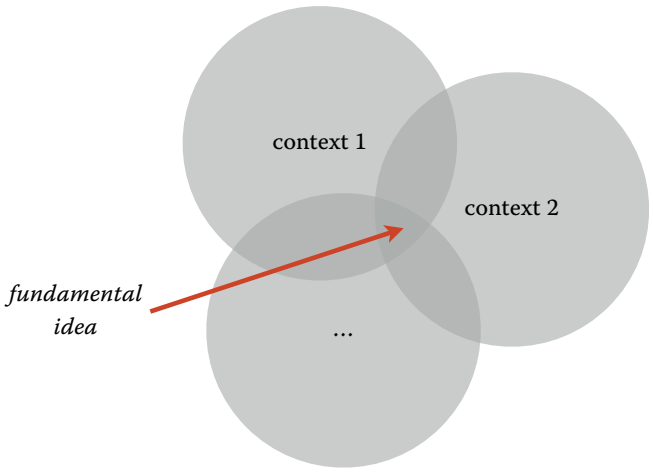


Figure 3.5 The criterion of sense prevails on the intersection of every day contexts.

Example 3.4 *Divide-and-conquer* Quicksort is an algorithm performing sort on a list of comparable values. The algorithm itself consists of instructions on how to perform sorting. Although sorting can be done in a way like the algorithm prescribes, it is based on a more fundamental principle: that of *divide-and-conquer*. It is this principle that can be used by many in every day life. Family members can for example divide cleaning tasks in and round home, merging the results. At work projects are divided amongst a group of people, to finally conquer it together. Even children building a sandcastle in a sandbox can divide tasks like 'filling buckets with sand' or 'building the west wing'. We

4 The *intersection* in mathematical terms.

see that divide-and-conquer satisfies the criterion of sense and therefore is a candidate for a fundamental idea.

3.2.4 *Criterion of time*

The last of the four criteria is the criterion of time. This criterion is important in two ways. First, a concept which is central in the development of a domain, is a good candidate for a fundamental idea. So observing the historical evolution of a domain, can give us a clue on how to find fundamental ideas (Schwill, 1994). Second, because the notion was important in history, it probably will be in the future.

> *Other things being equal, ideas that have impressed our predecessors are more likely to continue to impress our successors than our latest discoveries will.* (Nievergelt, 1990, page 5)

Figure 3.6 visualises these ideas with a time line and a fundamental idea which connects different points in history as well as in the feature.

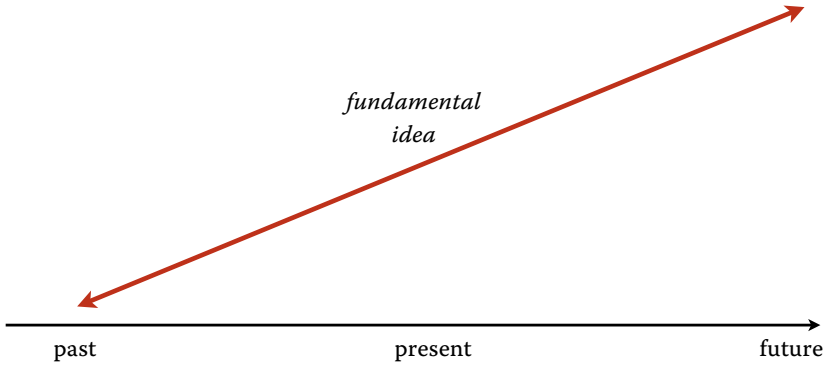


Figure 3.6 A fundamental idea should have importance in history and probably will stay important in the future.

Example 3.5 *Language* The notion of *language* has a long history in computer science. Grammars and syntax of formal languages where studied by Chomsky and Kleene in the 1950s. To let computers use languages, Knuth invented LR parsers in 1965. This was in 1969 extended to LALR parsers by DeRemer. Recursive descent parsers by Lucas and LL parsers by Lewis and Stearns are other examples. Studies to the semantics of formal languages were done by Hoare (axiomatic semantics), Scott (denotational semantics) and Manna (operational semantics) around the 1970s. All these developments in formal languages suggest this notion is in the centre of computer science. Therefore it is a good candidate for a fundamental idea.

3.2.5 *Overview*

Collecting all thoughts from last sections we can present a definition for a fundamental idea, as stated by Schwill (1994):

Definition 3.6 ***Fundamental idea*** A *fundamental idea* is a schema for thinking, acting, describing or explaining fulfilling four criteria:

- *Criterion of width*
  The idea is applicable or observable in multiple ways in different areas of the domain.
- *Criterion of depth*
  The idea may be demonstrated and taught at every intellectual level.
- *Criterion of sense*
  The idea is related to everyday language and/or thinking.
- *Criterion of time*
  The idea can be clearly observed in the historical development of the domain and will be relevant in the longer term.

3.3 The quest for fundamental ideas

Now that we know what fundamental ideas are and which criteria they have to satisfy, we can start our search for fundamental ideas of computer science. In the next sections, we present three alternative ways to look for fundamental ideas in our discipline. Two of them are theoretical of nature, one is empirically grounded. At the end of the section we shortly discuss the differences between and relevance of each approach.

3.3.1 *Cross-computing tools and techniques*

In Example 3.2 we encountered a categorisation of computer science by the ACM. Next to a conceptual categorisation, the ACM also acknowledges some '*cross-computing tools and techniques*' appearing in all fields.[5] They are listed in Box 3.1. We can see the tools and techniques are strongly influenced by engineering.

Box 3.1 Cross-computing tools and techniques by ACM (2012)

- Reliability
- Empirical studies
- Measurement
- Metrics
- Evaluation
- Experimentation
- Estimation
- Design
- Performance
- Validation
- Verification

5 The *Computing Classification System* can be found on-line at *http://www.acm.org/about/class/class/*.

3.3.2 *Catalogue of fundamental ideas*

Schwill (1994) does not only formalize Bruner's notion of fundamental ideas, he also proposes a procedure of four steps to obtain a set of fundamental ideas which satisfy his four criteria. These steps are:

I Analyse the concrete contents of a science and determine relationships and analogies between its subjects (criterion of width) as well as between different intellectual levels (criterion of depth).
II Revise and improve this set by checking whether each idea has a meaning and can be found in everyday life (criterion of sense).
III Try to review the historical development of each idea (criterion of time).
IV Tune the set of ideas according to the following questions:
- Do the ideas have a similar level of abstraction?
- Is it possible to structure or group the ideas somehow?
- Are there any hierarchical or network dependencies between them?
- Are the ideas linearly independent?

Schwill initiates the above procedure with the analysis of the software development process, because it is '*a central purpose of computer science*' (Schwill, 1994, page 283). After revising and tuning he comes to a set of fundamental ideas as presented in Box 3.2. The fundamental ideas are hierarchically structured and categorized in three *master ideas*: *algorithmisation*, *structural dissection* and *language*.

Algorithmisation is breaking processes down into single steps (Schwill, 1994, page 288). Dividing a system into components and determining their interrelations, however, is called structured dissection. An example is writing down a problem analysis in a precise and structured way as opposed to writing down the procedure to do such an analysis, which is algorithmisation. The divide-and-conquer approach, for instance, contains both an algorithmic and a dissection aspect. It is assigned to algorithmisation since the dynamic aspect (process aspect) prevails. On the contrary, dissection stresses the static aspect, i.e. the result of the dissection process and not the way the result is achieved.

Because formal languages return in almost all facets of computer science, these ideas are in a separate category. Not only programming languages, but also specification languages, query languages, command languages and logic come to mind.

Furthermore several ideas occur multiple times in different contexts in the catalogue. Reduction and transformation, for instance, denote translation processes, translation on the other hand appears again as an idea for implementing hierarchies. So it can be seen that ideas are intertwined in many ways, an exact separation and assignment is hardly possible.

3.3.3 *Central concepts*

The lists of tools, techniques and fundamental ideas from the previous two sections, have one main downside: they relay on the judgement of one or just a few

Box 3.2 Catalogue of fundamental ideas by Schwill (1994). Names written between parentheses are added by Schwill to denote groups and are not meant as fundamental ideas themselves. All fundamental ideas are categorized in three '*master ideas*': algorithmisation, structural dissection and language.

Algorithmisation

- (Design paradigms)
  - Branch-and-bound
  - Divide-and-conquer
  - Greedy-approach
  - Plane-sweep
  - Backtracking
- (Programming concepts)
  - Concatenation
  - Alternation
  - Iteration
  - Recursion
  - Non-determinism
  - Parametrisation
- Process
  - Concurrency
  - Processor
- Verification
  - Correctness
  - Termination
  - Consistency
  - Completeness
  - Fairness
- Complexity
  - Reduction
  - Diagonalisation
  - Order
  - Unit-cost measure
  - Log-cost measure
  - Worst/best/average/amortized case

Structured Dissection

- Modularisation
  - Top-down method
  - Bottom-up method
  - Information hiding
  - Locality of objects
  - Specification
  - Abstract data type
  - Team work
- Hierarchisation
  - Nesting
  - Tree
  - Parentheses
  - Indentation
  - Translation
  - Interpretation
  - Operational extension
- Orthogonalisation
  - Emulation

Language

- Syntax
  - Accepting
  - Generating
- Semantics
  - Consistency
  - Completeness
  - Transformation

experts. Zendler and Spannagel advocate the need of a systematic and empirical investigation. Instead of looking for fundamental ideas from theory, they set up an empirical study where the list of fundamental ideas is based on ratings of numerous experts. They surveyed experts in computer science to rank fundamental ideas and formed a catalogue based on their oppinion.

Zendler and Spannagel filtered 49 concepts from the *Computing Classification System*, discussed in §3.3.1. Only concepts which are mentioned more than ten times were taken into account. Thereafter, professors from fourteen universities in Germany were asked to fill in a questionnaire. This questionnaire consisted of four sections each related to one criterion (width, depth, sense and time) and each section contained questions about the same 49 concepts of computer science. The subjects were asked to rate their agreement to the concept satisfying each criterion on a five-point Likert scale. The result, determined by using clustering techniques, is a list of 15 *central concepts* of computer science. They are important in computer science education regarding the four criteria. These are listed in Box 3.3.

Box 3.3 Central concepts by Zendler and Spannagel (2008).

- Problem
- Data
- Computer
- Test
- Algorithm
- Process
- System
- Information
- Language
- Communication
- Software
- Program
- Computation
- Structure
- Model

3.3.4 *Overview*

We have three lists of fundamental ideas, each acquired with a different approach. These three separate lists however, complement each other. The ACM *Computing Classification System* contain *cross-computing tools and techniques* and a *classification* of the discipline in 12 fields. The *tools and techniques* are fairly engineering orientated and indeed cross-cutting through the discipline, they can directly be used as fundamental ideas. The *classification* is used by Zendler and Spannagel (2008) as a base for their study on fundamental ideas of computer science. They let a panel of experts check which terms satisfy the criteria of Schwill (1994). Schwill himself created a procedure, incorporating his own criteria, to produce a list of fundamental ideas. By combining these three approaches we gain a list of fundamental ideas from an engineering perspective, an empirical approach and a computer science theory viewpoint. Table 3.1 summarises all three approaches.

Table 3.1 Overview of approaches to fundamental ideas by the three authors discussed in this section. Two of the three approaches are theoretical based, one is empirical. Two are based on the *Computing Classification System* (CCS), one is entirely based on own theory.

Author	Name	Approach	Source
ACM (1998)	Cross-computing tools and techniques	theoretical	CCS
Schwill (1994)	Fundamental ideas	theoretical	–
Zendler and Spannagel (2008)	Central concepts	empirical	CCS

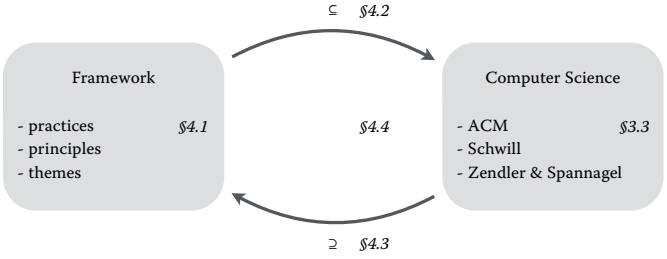
4

Towards a framework for Computing, Science, Technology, Engineering and Mathematics

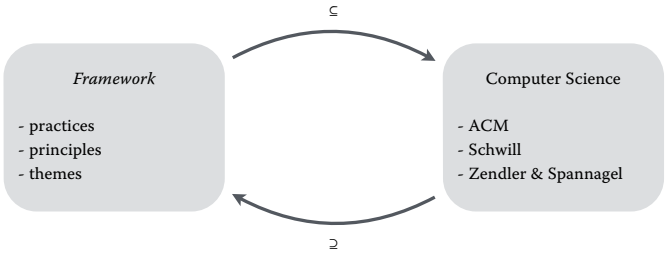
In Chapter 3 we have analysed a different method to structure computer science, based on fundamental ideas. It is a method orthogonal to the fields described in Chapter 2. In this chapter we develop a conceptual framework which incorporates fundamental ideas of computer science and can help to build computer science curricula for middle and high school. This framework will be based on 'A Framework for K-12 Science Education' by The National Research Council (2012). By doing so, we create a framework which can be used to design a curriculum for any STEM *subject, including computer science, rooted in fundamental ideas. Questions that arise are:*

- *How do the components of the framework for* K-12 *science education fit computer science?*
- *Where do the fundamental ideas obtained in §3.3 fit into this framework?*
- *How can we extend the existing* K-12 *science education framework to include computer science fundamental ideas?*

After we have introduced the framework by The National Research Council in §4.1, this chapter consists of two main parts. First we study the applicability of the existing framework to computer science in §4.2, discussing which parts are relevant to computing and why. Then we change perspective in §4.3, and check which fundamental ideas from §3.3 fit into the framework. After a review of our findings we propose some modifications to the framework to better fit in computer science in §4.4. Graphically we can represent the structure of this chapter in the following diagram.



4.1 A framework for K-12 science education



In this section we examine the overall structure of *A Framework for K-12 Science Education* by The National Research Council and discuss the usage of fundamental ideas in the framework.

4.1.1 *Goals and foundations*

The American National Research Council appointed a committee to design a conceptual framework for new K-12 science education standards. The goal of this framework is multiple fold. As stated by The National Research Council (2012), at the end of 12th grade, all students:

- Should have some appreciation of the beauty and wonder of science.
- Possess sufficient knowledge of science and engineering to engage in public discussions on related issues.
- Are careful consumers of scientific and technological information related to their everyday lives.
- Are able to continue to learn about science outside school.
- Have the skills to enter careers of their choice, including (but not limited to) careers in science, engineering, and technology.

4.1.2 *Three dimensions*

The framework is composed of three dimensions. The dimensions outline knowledge in science and engineering every student should learn at the end of high school. The National Research Council stresses all dimensions should be used together in standards, curricula, instructions and assessments to support students learning. The three dimensions are:

D1 *Practices* of science and engineering.
D2 *Principles* applicable across all STEM disciplines.
D3 *Themes* for every STEM discipline.[1]

1 The National Research Council (2012) speaks of *cross-cutting concepts* instead of *principles* and *core ideas* instead of *themes*. Because we divided knowledge in concepts, skills and attitudes before, we like to make a clear distinction between *concepts* as defined in §2.1 and the *principles*

Practices and principles are cross-cutting through all STEM subjects, but themes differ by discipline. A schematic overview of the framework and its dimensions is given in Figure 4.1. We discuss each of these dimensions shortly in the following sections.

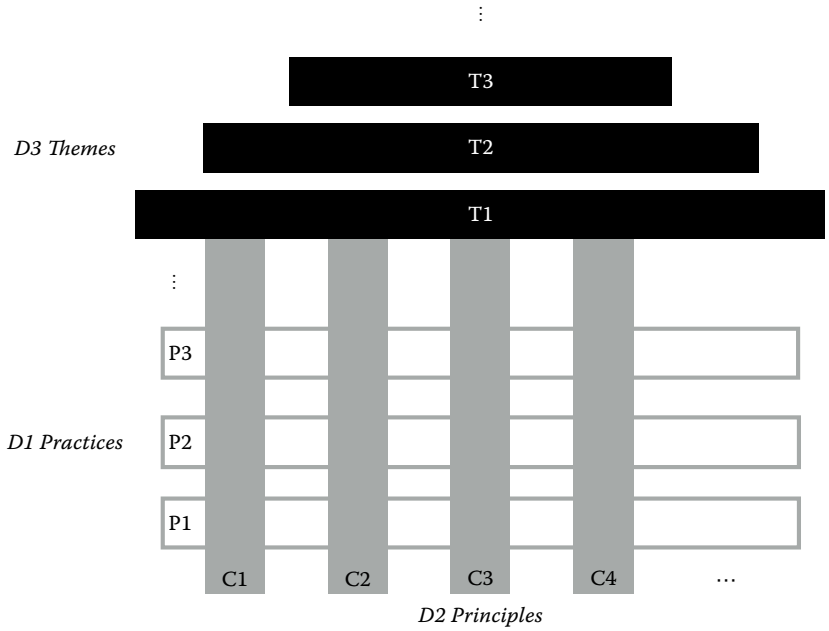


FIGURE 4.1 The structure of the framework developed in this document. *Principles* form a foundation for every STEM discipline. The *practices* are cross-cutting through each practice and supports the foundation. Every discipline is based on *themes* which are build upon the shared practices and principles.

DIMENSION 1 *Practices* Practices are the pillars on which STEM subjects in middle and high school should be build[2] as drawn in Figure 4.1. All practices are drawn from major methods and skills scientists and engineers use in their daily business. Students are encouraged to immerse themselves in these practices and to explore why they are so important in science and engineering. Each practice comes with major competencies a student should have at the end of 12^{th} grade. These *goals* and the way students competence level *progresses* during preceding grades, can be found in the original document. The eight practices introduced by The National Research Council (2012) are:

P1 Asking questions and defining problems
P2 Developing and using models
P3 Planning and carrying out investigations

introduced in this framework. Also, we want to make a distinction between *fundamental ideas* as defined in §3.2 and *core ideas*. Therefore we prefer to use the terms *principles* and *themes* instead of *cross-cutting concepts* and *core ideas*.

2 And thus the curricula build on this framework.

P4 Analysing and interpreting data
P5 Using mathematics and computational thinking
P6 Constructing explanations and designing solutions
P7 Engaging in argument from evidence
P8 Obtaining, evaluating, and communicating information

With respect to the practices, The National Research Council (2012) makes two remarks. First is the usages of the term *practices* instead of *skills*. This way the authors want to emphasize that engaging in scientific investigation and engineering inventions requires not only skill but also knowledge. This knowledge is specific to each practice. Second is that in doing science or engineering, all practices are used *iteratively* and *in combination*. The practices should *not* be used as linear and, in particular, *not* sequential in the way they are presented.

Dimension 2 *Principles* Principles are intertwined in every practice. They hold the pillars formed by the practices together (see also Figure 4.1) and support all themes from the third dimension. The application of these seven principles is across all domains of science:

C1 Patterns
C2 Cause and effect
C3 Scale, proportion, and quantity
C4 Systems and system models
C5 Flows, cycles and conservation of energy and matter
C6 Structure and function
C7 Stability and change

As intended by the criterion of width, these concepts help students to organize and connect knowledge from various disciplines into a coherent cognitive structure. The authors stress their role in the development of standards, curricula, instructions and assessments.

Dimension 3 *Themes* The third dimension consists of themes in stem, grouped by four disciplines:

- Physical sciences
- Life sciences
- Earth and space sciences
- Engineering, technology and application of science

A curriculum for a stem subject combines cross-cutting practices, cross-cutting principles and disciplinary themes. Themes covered by by above disciplines are listed in Box 4.1. In general themes should satisfy four points:

I Have broad importance across multiple sciences or engineering disciplines or be a key organizing principle of a single discipline.
II Provide a key tool for understanding or investigating more complex ideas and solving problems.

III Relate to the interests and life experiences of students or be connected to societal or personal concerns that require scientific or technological knowledge.
IV Be teachable and learnable over multiple grades at increasing levels of depth and sophistication. That is, the idea can be made accessible to younger students but is broad enough to sustain continued investigation over years.

BOX 4.1 Themes for each of the four disciplines covered by the original framework (The National Research Council, 2012).

Physical Sciences

T1 Matter and its interactions
T2 Motion and stability: Forces and interactions
T3 Energy
T4 Waves and their applications in technologies for information transfer

Life Sciences

T5 From molecules to organisms: Structures and processes
T6 Ecosystems: Interactions, energy, and dynamics
T7 Heredity: Inheritance and variation of traits
T8 Biological evolution: Unity and diversity

Earth and Space Sciences

T9 Earth's place in the universe
T10 Earth's systems
T11 Earth and human activity

Engineering, Technology, and Applications of Science

T12 Engineering design
T13 Links among engineering, technology, science, and society

4.1.3 *Fundamental ideas*

The framework developed goes well with the message of fundamental ideas. The authors articulate their point of view as follows:

> *...the committee concludes that* K-12 *science and engineering education should focus on a limited number of disciplinary themes and cross-cutting concepts, be designed so that students continually build on and revise their knowledge*

> *and abilities over multiple years, and support the integration of such knowledge and abilities with the practices needed to engage in scientific inquiry and engineering design.* (The National Research Council, 2012, page 2)

Two main points present itself. The first is that the authors created a framework based on a '*limited number of disciplinary themes and cross-cutting concepts*'. Meaning the framework is build on some sort of fundamental ideas per discipline and fundamental ideas that fit each STEM subject. The second striking similarity is the goal the authors envision by using themes and cross-cutting concepts: '*so that students continually build on and revise their knowledge and abilities over multiple years*'. This is exactly the main benefit Bruner (1960) advocates, as we already discussed in §3.1.1.

Also, the criteria disciplinary themes should adhere, as listed on page 42, are in agreement with the criteria Schwill (1994) conceived for fundamental ideas. First, point I states exactly the same as the criterion of width. Both point II and point IV are implied by the criterion of depth. And at last, the criterion of sense is equivalent to point III. Only the criterion of time is not reflected in above points.

4.1.4 *Place of computer science*

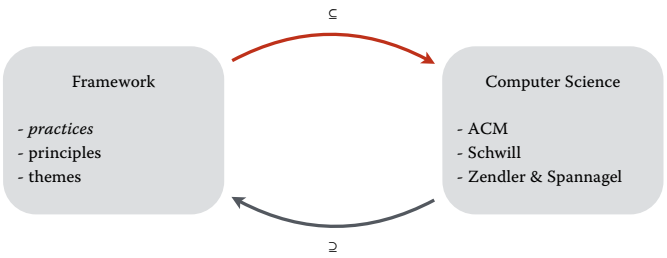
Notably absent from the disciplines covered by *A Framework for K-12 Science Education* is computer science. The National Research Council does not address computer science, arguing it can be '*seen as a branch of the mathematical sciences, as well as having some elements of engineering*' and having '*a history and a teaching corps that are generally distinct from those of the sciences*' (The National Research Council, 2012, page 15). Therefore the committee has not taken integrating computer science into the framework as part of their charge.[3] The authors stress, however, computer science should not be excluded from the K-12 curriculum and certainly has a place in K-12 education. The notion of *computational thinking* is mentioned as a practice though, recognising the need of computing in a modern STEM framework.

4.2 Applicability of practices and principles to computer science

The first step in adapting *A Framework for K-12 Science Education* to computer science, is to examine the applicability of the framework for this discipline. Before making any changes to the framework, we step over each cross-cutting *practice* and cross-cutting *principle* to check in which way they can be used in computer science.

3 Statistics is another discipline not included in the framework.

4.2.1 *Cross-cutting practises*



All practices apply to both scientists as well as engineers in general. Some notions however, are used differently by scientists and engineers. Science is focused on investigating and creating models and theories about the world. Engineering instead focuses on designing and building systems. As computer science is regarded as a scientific and as an engineering discipline, both viewpoints apply equally well (Tedre, 2014). The practices discussed in the next sections are therefore easily applicable to computer science. Students in middle and high school are most likely to come into contact with the creation and developing part of computer science. Hence we discuss each practice focussing on the engineering viewpoint to computer science.

Practice 1 *Asking questions and defining problems* The development cycle of a software project usually starts with a *problem*, need or desire. As in engineering, this problem needs to be pinpointed, narrowed down and described in a clear and well defined way using a *specification*. Software developers ask *questions* to narrow down the class of the problem, determine criteria a solution has to satisfy and write all these constraints down.

Example 4.1 *Non-linearity of practices* We use this practice as an example to emphasize the fact that practices are not linear, as mentioned before at page 41. Developers continually ask questions to help them and their team in defining new (sub)problems, change perspective on current work and control their next steps. During the design phase of a new product, multiple solutions are compared and evaluated (see also Practice 7). When creating an implementation, a developer will ask questions like 'Will it meet the design criteria?', 'Can multiple ideas be combined to produce a better solution?', 'Are ther possible trade-offs of this implementation?' and so on. Also in the time of testing, a product faces numerous quality assurance questions. Some of them it will abide, others will reveal bugs or incompletenesses.

Practice 2 *Developing and using models* Software developers make extensively use of *models* and *simulations* to analyse data flow and program flow. Programs are written in a paradigm, such as object orientated, functional orientated, aspect orientated and many more. Choosing for a particular paradigm or language, or using a specific framework gives a programmer tools to model a problem for a device.

At the same time such a choice constrains the developer in the way he/she can design a solution. Software developers have to, like scientists, commit to a paradigm and use the concepts common to it. But they can also explore paradigms at a meta level, extend them or find a new ones. Paradigms and models are never complete and can always be changed and updated. Developers must be aware of intrinsic limitations of models, paradigms, languages and APIs they use. *Abstracting* from the real world in a model and *instantiating* a model to a real world object are central to this practice.

Practice 3 *Planning and carrying out investigations* Next to creating and using models or producing an explanation or artifact, computer scientists use *experimentation* and *measurement* to gain data for different purposes. One such purpose could be during the initialization of a project, when investigating the requirements of a product. Another is during the design phase, where investigations help to determine how effective, efficient, durable, user friendly and so on a design may be. In software development, and also engineering in general, this is often called *quality assurance*. Thus investigations to gain data are essential for both specifying design criteria or parameters, as well as testing prototypes and products.

The usage of this practice in engineering is mainly to test designs. This fact reminds of *verification*, *validation*, *correctness*, *consistency*, *completeness*, *fairness* and *testing*. These terms, however, are not directly mentioned by the authors of the framework. We like to make these fundamental ideas more visible in this practice. For now, we just check how this practice is applicable to computer science. Therefore we like to delay these extensions to §4.4.1.

Practice 4 *Analysing and interpreting data* A great part of computer science is about data: data analysis, data storage, data communication, data manipulation... However, that is a component specific to computer science and should be a theme connected within the discipline. We will come back to this later, in §4.4.2. This practice is about ordering collected data during investigations (see also Practice 3) to reveal patterns, find explanations and give answers. In doing so, one needs a form to visualize these patterns and relationships and allows the result to be presented to others.

The intended meaning of data by this practice is thus data collected during requirements engineering, testing, prototyping and other *empirical studies* done by software developers and engineers in general. This data is used to compare different solutions to a problem and determine how well solutions meet specified criteria. Engineers like to know '*which design best solves the problem within the given constraints*' (The National Research Council, 2012, page 51). They make decisions based on the result of such an *evaluation* and use it as evidence that a given solution will work, instead of falling back on trail and error. These *metrics* can be used to make informed design decisions and as *estimation*s of performance and quality after delivery. It helps clarifying problems, determine economic feasibility, evaluate possible alternatives and investigate possible failures. Analysis can be done using a

model (see also Practice 2) or by creating prototypes. Data can be gathered through testing under different conditions.

Practice 5 *Using mathematics and computational thinking* As in every STEM subject, mathematics is really important in computer science. Computer science even originated as a mathematics discipline (Tedre, 2014). Of course the inclusion of computational thinking in a practice emphasizes the importance of computer science as a school subject. Mathematical and computational thinking can be developed inside every STEM subject, but the theoretical foundations have to be developed in their own subjects. For mathematical thinking these will be courses containing calculus, statistics, linear algebra and geometry. For computational thinking these will be courses about programming, algorithms, data and computer devices.

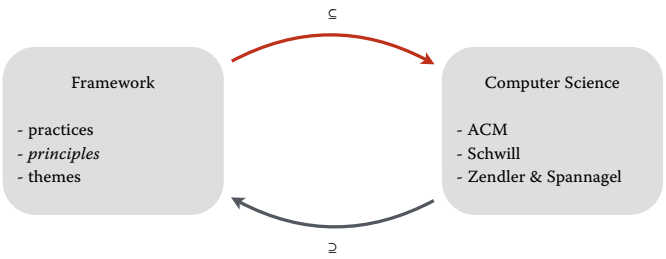
Practice 6 *Constructing explanations and designing solutions* The first part of this practice, '*construction explanations*', is directed towards science. Scientists like to construct theories to explain properties of the world. This explanation should be consistent with available evidence and utilized theories. The second part, '*designing solutions*', is directed towards engineers. Computer scientists and software engineers give explanations to test results during verification processes: explaining quality aspects like time and space complexity, usability and so on. The main concern of an (software) engineer is *designing* solutions, solutions to problems society, business, science or organisations ask for.

Practice 7 *Engaging in argument from evidence* Reasoning and argumentation are essential in finding the best explanation of a phenomenon or the best solutions to a problem. For example, in an initial stage, software engineers have to compare multiple design ideas. Later on in the design process, they test prototypes and collect data (see also Practice 8). This data is used as evidence in argumentation about the strengths and weaknesses of the proposed designs. Without argumentation scientists and engineers could not discuss and examine their explanations and solutions. Next to systematic methods to compare alternatives, usage of test data to formulate evidence and so on, there is also an epistemic argument supporting the importance of this practice:

> *The knowledge and ability to detect "bad science" are requirements both for the scientist and the citizen.*
>
> (The National Research Council, 2012, page 71)

Practice 8 *Obtaining, evaluating, and communicating information* Like scientists, software developers have to obtain information form various sources. It is not enough to find information and use it without thinking. As stated above, evaluation of the gathered information is crucial. Controlling the source and having a sense of rightness of the information is important. Epistemic knowledge plays a crucial role in this practices.

4.2.2 *Cross-cutting principles*



The second dimension of the framework consists of cross-cutting principles. These principles are as important as the practices in each of the STEM disciplines. As with the practices we discuss shortly the applicability of each principle to computer science in the next sections.

Principle 1 *Patterns* Patterns are everywhere in science, engineering and computer science. Recognising, ordering and using patterns is a core business of every computer scientist and software engineer during the whole software development cycle, including definition, designing and verifying. *Pattern matching* is used by inductive definitions used by many mathematical theories in computer science. It is also made explicit in many (functional) programming languages.

Principle 2 *Cause and effect* Every effect or result produced by an information system is caused the exact instructions a computer has been told to do. The goal of engineers is to design a solution which causes a desired effect. The understanding of the causal relationships of a model or design will help students to explain and reason about the system. Most of the time it is not the case that 'the computer does not do what you want it to do,' but 'it does exactly as you told it to do' and you probably told it to do the wrong thing. Cause and effect are essential principles when modelling systems, writing and debugging programs and other common scientific and engineering practices a computer scientist uses.

Principle 3 *Scale, proportion, and quantity* This principle is important in software development when, for example, scaling up a solution to address more connections, customers, data and so on. Also this concept applies very well in complexity theory when analysing *worst case*, *average case*, *best case* and *amortized case* of runtime and data usage.

Principle 4 *Systems and system models* Information *systems* are a main object of study for computer scientists. Everything from a chip to a computer (hardware) and from a script to an operating system (software) can be regarded as a system getting input, processing data and producing output. (See also Principle 5. Modelling is already discussed in Practice 2 at page 45.)

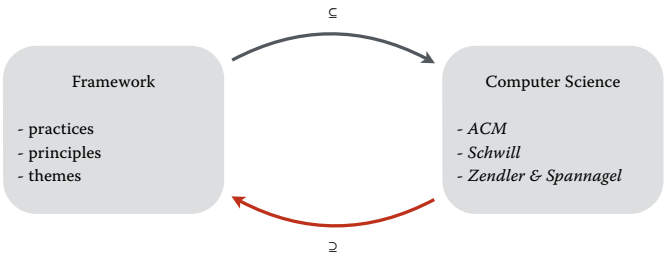
Principle 5 *Flows, cycles and conservation of energy and matter* Computer scientists are not concerned with matter and energy. Although energy consumption by computer systems and software is starting to become a research subject, we cannot say it is a main aspect of computer science as it is in all other science and engineering disciplines.

Flow of data on the other hand is core business of computer scientists. To reflect this idea we will modify this principle in §4.4.1.

Principle 6 *Structure and function* To illustrate this principle, we give a simple example: the *structure* of a program (the way it is build) has to reflect the *function* of it (the way it will be used). It is the same as the design of a bike: the structure of the frame has to reflect that it has to carry a person and be mounted to two wheels. When developing a program, software engineers continually need to evaluate the structure of their code to meet the function of the solution.

Principle 7 *Stability and change* Some computer systems have to be stable, serving information to a lot of people for instance. Others have to change rapidly, like altering the amount of products in a warehouse. This concept is as important in computer science as it is in engineering in general.

4.3 Applicability of fundamental ideas to practices and principles



Our main concern in this section is taking a look at the fundamental ideas from §3.3 and relate them to the practices and principles from the framework. The fundamental ideas we link here are general enough to be applicable to all STEM disciplines. They can be removed from our 'check-list' and do not have to appear in the disciplinary themes on computer science. As the reader gets along, he/she can keep a record of the fundamental ideas already included in the original framework. To ease this tracking of fundamental ideas throughout the chapter, we *emphasise* their occurrences. All fundamental ideas can be found in Appendix A.

List 1 ACM The ACM tools and techniques are strongly orientated towards engineering. It is almost trivial to look for practices and principles that incorporate these ideas. The ideas of *empirical studies*, *experimentation* and *measurement* can be found in Practice 3 and Practice 4. *Evaluation*, *metrics* and *estimation* are part of Practice 4. *Design* is off course the base of Practice 6. As stated before, ideas like *reliability*, *performance*, *validation* and *verification* are good candidates to be

added to Practice 3. We come back to this in §4.4.1, where we will put forward some modifications to the framework.

List 2 *Schwill* The fundamental ideas listed by Schwill are fairly algorithmic in nature and most of them are not general enough to be covered by a cross-cutting practice or principle. *Structured dissection* however is covered by Principle 1 and Principle 6. The ideas of *hierarchisation*, *nesting*, *tree*, *parentheses* are *indentation* part hereof. The same applies to *orthogonalisation*, *emulation*, *modularisation* and *top-down method* and *bottom-up method*, which can also be found throughout both practices. Ideas about *complexity* like *worst case*, *average case*, *best case* and *amortized case* of runtime and data usage can be found in Principle 3. A fundamental idea of Schwill which clearly covers engineering is *specification*.

List 3 *Zendler and Spannagel* The idea *problem* is in the top five of the list Zendler and Spannagel (2008) put together in their empirical search for fundamental ideas. A *question* is on the same line as a problem, but mainly focused towards science.[4] Both are part of Practice 1. Practice 2 covers *model*.

We can find some other fundamental ideas mentioned by Zendler and Spannagel in the principles of the framework. For example, a *system* is studied in Principle 4 and a *process* is part of Principle 5. The idea of *structure* manifests itself in Principle 6. Once more we do not have a evident place to classify *test*. Practice 3 would be a good candidate, but is not specific enough towards testing and verification.

4.4 Embedding remaining fundamental ideas of computer science

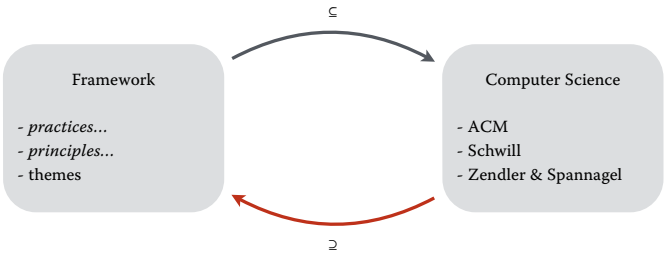
In this section we divide the remaining fundamental ideas in two classes. The first class contains ideas which can be applicable to all stem disciplines, including computer science. Therefore they should be embedded in already existing practices or principles. We will present some modifications compared to the original k-12 science education framework discussed in the sections above. Of course, when extending original practices or practices, we have to be sure this extensions are relevant to *all* stem subjects. In §4.4.1 we extend two practices (Practice 3 and Practice 4) and alter one principle (Principle 5) to better reflect cross-cutting fundamental ideas found in Chapter 3. Next to that, we add one new practice.

The second class contains ideas which are not applicable to all stem subject. That means these fundamental ideas are *pure computer science orientated*. They will not fit into cross-cutting practices or principles and we have to fit them into disciplinary themes for computer science. In §4.4.2 we define three disciplinary themes for computer science which incorporate all left over fundamental ideas.

In expanding the cross-cutting dimensions and defining the disciplinary themes, we also incorporate our results from Chapter 2 by enacting the focus of the curricula and guidelines in our additions.

4 The term *question* was not an object of study for Zendler and Spannagel (2008).

4.4.1 *Cross-cutting fundamental ideas*



Because of the relevance of verification and structuring in computer science, we like to modify two practices from the original framework: P3 Planning and carrying out investigations, and P4 Analysing and interpreting data. The value of collaboration is demonstrated by the CSTA curriculum guidelines and the fundamental idea *team work* by Schwill. To reflect this importance, as well to computer science as to all other STEM subjects, we add one more cross-cutting practice: P9 Collaborating with peers and experts. The non-applicability of the principle C7 (Flows, cycles and conservation of energy and matter) to computer science, can be fixed by slightly generalising the topic and adding *data* to energy and matter. All modifications and additions are summarized in Table 4.1. In the next sections we will discuss these four adjustments one by one.

Table 4.1 Modifications to the original framework as defined by The National Research Council (2012). Old names of practices and principles are listed (if applicable) with the new names after the arrow.

P3	Planning and carrying out investigations
	⇝ Planning and carrying out investigations *and verifications*
P4	Analysing and interpreting data
	⇝ *Structuring,* analysing and interpreting data
P9	–
	⇝ Collaborating with peers and experts
C5	Flows, cycles and conservation of energy and matter
	⇝ Flows, cycles and conservation

Practice 3 *Planning and carrying out investigations* and verifications An important objective for software developers is to test or *verify* their products against requirements and specifications. This weightiness is reflected by the *V-model* (IEEE, 2011), which assigns one of its two branches to *verification*, making it as important as *creation*. To stress the importance of verification, we change the name of this practice from '*Planning and carrying out investigations*' to '*Planning and carrying out investigations and verifications*'. At a glance, it should be clear *verification* and *validation* plays as big a role in this practice as is *investigation*. Even more so in engineering as in science. Other fundamental ideas about quality assurance fit in as well: *correctness*, *consistency*, *completeness*, *fairness*, *reliability*, *performance* and *test*ing.

We have to validate the addition of *verification* to this practice for every STEM subject. Where engineers speak of the 'verification of a product', scientists speak of 'verifying a hypothesis'. Engineers test prototypes or products for quality assurance. Scientists collect data to '*test existing theories and explanations or to revise and develop new ones*' (The National Research Council, 2012, page 50). Testing and verification are thus as applicable to computer science as to every other STEM discipline.

Practice 4 Structuring, *analysing and interpreting data* This practice is not just about analysis and interpretation. Before raw data can be analysed or interpreted, it has to be structured an organized in a way to reveal its information. Science and engineering use tools like tabulating, graphing, visualizing and statistical analysis to find the explanations and relationships between variables and concepts. Hence we like to add '*structuring*' to this practice, to express the necessity of ordering data before analysing or interpreting it.

To systematically analyse something, one can use different *modularisation* and *hierarchisation* techniques as discussed by Schwill (1994). Scientists and engineers can use a *top-down method* or a *bottom-up method* to structure their data or set up their research or design. To visualize data in a hierarchical way, one can use *nesting*, *trees*, *parentheses* and *indentation*.

Practice 9 *Collaborating with peers and experts* An important omission in the original framework is the lack of *collaboration* and *team work*. Although Practice 7 implies collaboration in science and engineering, the framework does not stress this directly. Schwill (1994) mentions *team work* as a fundamental idea. It is not a fundamental idea specific to computer science and it is general enough to be in one of the cross-cutting dimension of the framework. However, it can not be placed in any of the existing practices or principles. We regard collaboration is not sufficiently brought to the attention by the original framework. To put this right, we add the practice '*Collaborating with peers and experts*'.

Students should be taught to work together with peers, as a pair or in bigger teams, and with experts, within the same field as well as multidisciplinary. Computer artefacts, for example, are not created by one person. Groups of developers are working together to produce new software. Not only do these teams have a multi disciplinary character, they often consist of people with different levels of expertise. Communication between team members, giving feedback etcetera, are skills every student should be able to use during middle school, high school and beyond. Collaboration is mentioned as a central practice in 21st century skills (KNAW, 2012). Although collaboration manifests itself in software development, it is too a main part of science and engineering.

Principle 5 *Flows, cycles and conservation* Energy and matter are not objects of study for computer scientists. They do however study flow and cycles in a system,

but it is all about *data* instead of energy or matter. Therefore we alter this principle to reflect the importance of data in computer science in particular and science and engineering in general. This is also done by the Dutch incarnation of *A Framework for K-12 Science Education* in the section on technology (Ottevanger, et al., 2014). When speaking about data flows and cycles, the idea of a *process* comes into mind, as well as flows operating at the same time inside a system: *concurrency*. An overview of this altered principle is given in Figure 4.2

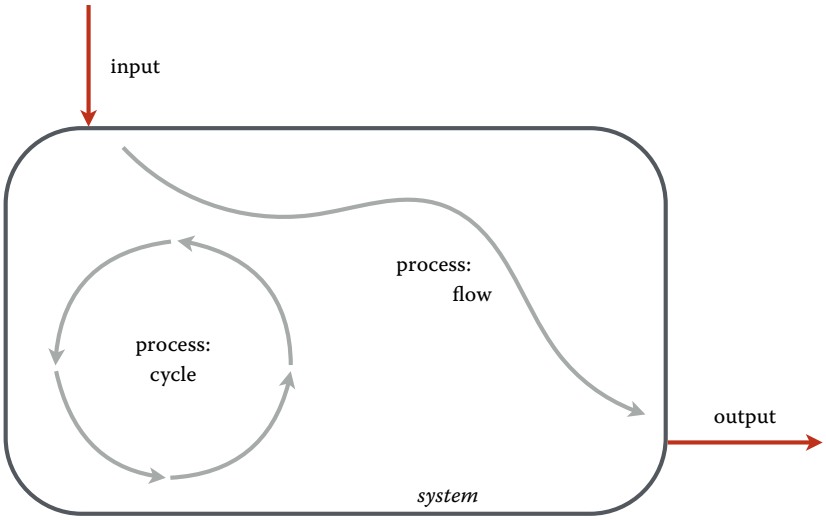
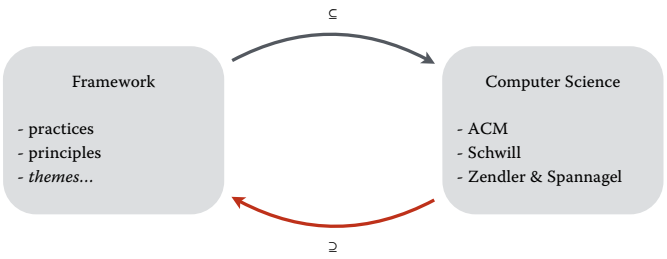


Figure 4.2 *Systems* are isolated parts of the universe. They receive an *input* and produce an *output*. *Flow* is energy, matter or data moving into the system, moving through the system and then moving out again. A *cycle* is when energy, matter or data is recycled over and over again within a system. The energy, matter or data inside a system is changing shape and form, it is being *processed*.

4.4.2 *Disciplinary fundamental ideas*



We have altered some practices and principles and we added a new one to embed cross-cutting fundamental ideas into the framework. We will incorporate the remaining fundamental ideas into the *disciplinary themes* on computer science. In formulating these themes, we also use our results from Chapter 2. The structure of the French and CAS curriculum and guidelines greatly help to assemble the remaining fundamental ideas. Analysing the unused fundamental ideas we come to these three disciplinary themes:

T14 Representation and interpretation of data
T15 Algorithms: their usage and implementation
T16 Digital devices and communication over networks

Below we discuss each disciplinary theme and show which remaining fundamental ideas from the lists in §3.3 they contain.

Theme 14 *Representation and interpretation of data* Computer science is about *data* and the *transformation* of data. Data is the input of an algorithm or program where it is processed and manipulated (Principle 5 at page 52). In the end the output of a system is also data, which should be *interpreted*. Humans and computers both have a very different view on data and *information*.

Data appears in numerous ways. For a computer to process data, it has to be in a specific form. Computer scientists make use of *abstract data type*s and *information hiding* to model information in a computer. Also the *representation* of data in a digital device or over a network is highly important. *Translating* data from and to its representation is done continually.

Theme 15 *Algorithms: their usage and implementation* Computer science is not just about data. Systems are made to *process* the data. The main model introduced in Principle 4 comes in hand. Computer programs are built to process input and produce output. An *algorithm* is used to manipulate data and perform a *computation*.

Algorithms, among others, are expressed in a *language*. Formal languages became a major research area of computer science. As physicists use mathematics to express themselves and their theories, computer scientists create their own formal language to express their systems and models. Languages are not only used for programming and making *software*. Different type of languages exist: specification languages, query languages, command languages and logic for example. Specification languages are used to formalize system requirements. Different command languages exist to express instructions for an operating system. To question a database computer scientists engineered query languages. This list can go on and on, growing much bigger. In designing an algorithm or program, students can use multiple design paradigms like *branch-and-bound*, *divide-and-conquer*, *greedy-approach*, *plane-sweep* and *backtracking*.

To study languages students have to understand the difference between *syntax* and *semantics*. Study *grammars* that *generating* and *accepting* languages, possibly using simple automata. This way students will learn the underpins of formal languages. Using fundamental ideas like *reduction*, *termination* and *diagonalisation*, students can be exposed to computation models.

Algorithmisation is the process of dividing a process into small steps. Students should be able to deduce and describe such steps and form an algorithm out of it using a language. During the creation of algorithms and *program*s, students need to know the building blocks they can use. These are *concatenation* (or *sequencing*),

alternation (or *branching* or *choosing*) and *repetition* in the form of *iteration*, *recursion* or *looping*. Next are notions of abstraction like *parametrisation* (using functions and procedures), *hierarchisation* and *modularisation*. Special techniques when developing algorithms are *non-determinism* and *concurrency*. Analysing the *complexity* of algorithms should be another major topic in this theme. Fundamental ideas like *order*, *unit-cost measure*, *log-cost measure* come to mind in this context.

Theme 16 *Digital devices and communication over networks* The development of software and the analysis of algorithms are not the only creative part of computer scientists. They also shape the world by creating *computer* and let these devices communicate with each other and their environment. Students should know the components of devices, most importantly the *processor*, which is the central unit processing data and performing algorithms. *Communication* between devices often happens over a *network*; *client-server* set-ups come to mind. The *locality of objects* and sharing of it is difficult to understand for students and demands great attention of teachers. The World Wide Web with all its technologies is of course a great example of a well known and very big network. Communication with the environment happens through sensors, displays, and many other *peripherals*. *Robotics* and *human computer interaction* can be gathered in this theme.

4.4.3 *Overview*

Our overall goal was to provide a conceptual framework to structure and design computer science curricula based on the notion of fundamental ideas. A curriculum based on such a framework gains all the benefits Bruner claims about fundamental ideas. These benefits were discussed in §3.1.1. We have adapted the original framework by extending the cross-cutting dimensions or by adding new themes for computer science to the disciplinary dimension. We have created a *conceptual framework for* K-12 *computing, science, technology, engineering and mathematics education*. In grouping these disciplines together we make *computing* a major part of it. To abbreviate this list of disciplines, we introduce the term CSTEM. The final list of the cross-cutting practices and principles and the disciplinary themes for computer science can be found in respectively Box 4.2 and Box 4.3 on the next pages.

BOX 4.2 Final list of practices, principles of the CSTEM framework. Practices and principles are cross-cutting through all CSTEM disciplines. Additions with respect to the original framework of The National Research Council (2012) are *emphasized*.

Practices

P1 Asking questions and defining problems
P2 Developing and using models
P3 Planning and carrying out investigations *and verifications*
P4 *Structuring,* analysing and interpreting data
P5 Using mathematics and computational thinking
P6 Constructing explanations and designing solutions
P7 Engaging in argument from evidence
P8 Obtaining, evaluating, and communicating information
P9 *Collaborating with peers and experts*

Principles

C1 Patterns
C2 Cause and effect
C3 Scale, proportion, and quantity
C4 Systems and system models
C5 *Flows, cycles and conservation*
C6 Structure and function
C7 Stability and change

BOX 4.3 Final list of themes of the CSTEM framework. Themes are disciplinary and grouped by area. Additions with respect to the original framework of The National Research Council (2012) are *emphasized*.

Physical Sciences

T1 Matter and its interactions
T2 Motion and stability: Forces and interactions
T3 Energy
T4 Waves and their applications in technologies for information transfer

Life Sciences

T5 From molecules to organisms: Structures and processes
T6 Ecosystems: Interactions, energy, and dynamics
T7 Heredity: Inheritance and variation of traits
T8 Biological evolution: Unity and diversity

Earth and Space Sciences

T9 Earth's place in the universe
T10 Earth's systems
T11 Earth and human activity

Engineering, Technology, and Applications of Science

T12 Engineering design
T13 Links among engineering, technology, science, and society

Computer Science

T14 *Representation and interpretation of data*
T15 *Algorithms: their usage and implementation*
T16 *Digital devices and communication over networks*

5

Conclusion

In this thesis we presented our contributions to computer science education in three parts.

I An analysis of the content knowledge of four international curricula and guidelines on computer science.
II A different viewpoint on computer science curricula using fundamental ideas, instead of a classical division in fields of the discipline.
III An implementation of fundamental ideas of computer science embedded in an existing conceptual framework for K-12STEM education.

Each part contributes in its own way to computer science education in (junior) high schools.

I The analysis of differences between and similarities of international curricula provides us with information useful on designing new curricula for computer science. It helps to reflect on topics taught in (junior) high school and their substance in a curriculum. In addition, it yields a categorised list of concepts in computer science, teachable to students in different stages of K-12 education.
II Fundamental ideas allow us to create a sustainable curriculum for computer science. Curricula based on fundamental ideas motivate students to extend their knowledge inside fields and across field borders, allowing for stronger cognitive structures and better non-specific transfer. We presented the criteria for fundamental ideas by Schwill (1994). Evaluation of theoretical and empirical studies produced a list of fundamental ideas that should be apparent in every (junior) high school curriculum on computer science.
III A framework which contains STEM subjects as well as computer science gives us opportunities to work interdisciplinary and better integrate common practices and principles. We show how computer science can fit in the existing framework *A Framework for K-12 Science Education* in two ways:
   – How the existing framework covers computer science: which cross-cutting practices and principles are applicable to computer science and why.

– How the fundamental ideas of computer science are apparent in the cross-cutting practices and principles.

After this investigation we extended the original framework by modifying two practices and one principle and by adding one practice. Next to this cross-cutting part, we developed three themes specific for computer science. These adjustments make sure all fundamental ideas of computer science are embedded in this CSTEM framework. Curricula based on this framework make sure to benefit from a wide, deep and sensible usage of fundamental ideas.

A

List of fundamental ideas

The next pages contain a list of all fundamental ideas mentioned in this document. The page references are to the place where they are defined in a theory, somewhere in Chapter 3 (pages 25–39), as well as the place where they fit into the framework, thus somewhere in Chapter 4 (pages 39–59).

B

List of categories and their concepts

Here we list all concepts filtered from the curricula and guidelines discussed in Chapter 2. Concepts are grouped by category, as presented in Table 2.2.

Algorithms

algorithm
algorithm representation
algorithm sharing
ambiguity
breadth first search
complexity
component
computationally unsolvable
concurrency
data processing
deadlock
decision
decomposition
depth first search
finite state machine
heuristic algorithm
information sharing
input
instruction
instruction sequence
instruction set
iteration
live lock
merge sort
output
parallel processing
parallel stream
parallelisation
pattern
performance
precision
problem solving
recursion
redundancy
repetition
resource
search algorithm
selection
sequence
sort algorithm
steps
task
tractability

Architecture

apis
architecture
assembly code
binary form

binary switch
bit
byte
chip
circuit
communication layer
compiler
computer
computer component
cpu
device
digital machine
digital value
electronic device
embedded system
emulator
execute
execution model
file io
file system
flip-flop
hand-held technology
hard disk
hardware
hardware component
hardware problem
hardware sharing
instruction representation
interpreter
logic circuit
logic gate
low level language
machine
memory
mobile device
monitor
moore's law
mouse
numeric value
operating system
overflow
peripheral
personal computer
physical layer
processor
real time system
register
sampling
scheduling
single event system
software
system
system controlling
system design
thread
translation
virtual machine
von neumann architecture

Data

array
audio format
big data
binary representation
character
character representation
compression
data
data error
data representation
data set
data storage
data type
data value
database management system
digital data
document format
file format
floating point
fraction
fraction representation
hexadecimal number
hexadecimal representation
image representation
information
information persistence
information representation

information system
integer
list (data structure)
lossless compression
lossy compression
mark-up language
number representation
persistence
query language
relational database
relational schema
representation purpose
representation sharing
retrieving information
sampling frequency
signed integer
sound representation
storage
string
table
text representation
two dimensional array
unsigned integer
word

Engineering

chart
clarity
collaboration
communication
correctness
data analysis
development instrument
digital artifact
documentation
evaluation
feedback
flowchart
functional design
implementation technique
instances
object-oriented design
problem
problem exploration
problem statement
productivity tool
project
project management
prototype
refactor
requirement
scale
separation
software creation
software development (process)
software life cycle process
solution
specification
stakeholder
teamwork
technology resource
technology tool
test case
tool
validation
verification
version control system

Graphics

image brightness
image contrast
image format
multimedia
multimedia tool
visual representation

Intelligence

artificial intelligence
computer vision
human intelligence
intelligent behaviour
language understanding
machine intelligence
robot component
robotics

Mathematics

and
binary number
boolean
exclusive-or
graph
logic
logical expression
not
or
quantization
set
statistical function
tree
truth table

Modelling

model
simulation

Networking

authentication
bandwidth
browser
client-server model
communication between machines
cookie
data communication
domain name service
error correction (network)
fault-tolerance (network)
firewall
http request
hyper link
internet
internet service
internet vs web
ip address
latency
mac address
mail header
network
network address
network component
network connection
network diagram
network functionality
network message
network path
network protection
network structure
network traffic
on-line resource
packet
packet switching
path (network)
peer-to-peer
point to point transmission
protocol
queue (network)
receiver
routing
search engine
search engine ranking
search query
server
server capability
shared resources (network)
spooler (network)
transmitter
url
web
web browser
web page
web page structure
web request
web site
web site address
web site name

Programming

application
argument (of function)
arithmetic operation

assignment
behaviour (of code)
boolean operation
bug
class
conditional
conditional jump
constant
context (of application)
control structure
data structure
divide by zero
efficiency
error
expression
function
high-level language
html
language
logical operation
looping (programming)
method
mobile computing application (programming)
paradigm
parameter
procedure
program
program creation
programming language
programming technique
readability (code)
recursive function
scope
semantic error
signature
statement
string manipulation
syntactic error
usability (code)
variable

Rest

business
information flow (business)
organisation structure (business)
project structure
standard

Security

access rights
cryptography
encryption
password
protection
secure storage
secure transaction
security
web safety and security

Society

appropriateness
bias
career
commercial software
comprehensiveness
digital rights
ethical behaviour
experience
expression (communication)
free software
hacking
information right
interdisciplinary
international network (society)
law
legal behaviour
limitation of digital machines
open source development
open source software
ownership (privacy)
personal information
privacy

productivity technology
proprietary software
public domain software
relevance
software license
software piracy
technology

Usability

adaptability
human computer interaction
user
user dialogue

References

ACM (1998). *Computing Classification System*. (Tech. Rep.). ACM. Available from http://www.acm.org/about/class/1998

_____ (2012). *Computing Classification System*. (Tech. Rep.). ACM. Available from http://www.acm.org/about/class/class/2012

ACM/IEEE (2013). *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. (pp. 1–172). Author. doi:10.1145/2534860

Académie des Sciences (2013). *Teaching computer science in France: Tomorrow can't wait*. (Tech. Rep., no. May). Paris: Académie des Sciences.

Barendsen, E., Demo, B., Grgurina, N., Izu, C., Mannila, L., Mirolo, C., ... Stupurienė, G. (2015). Key Concepts in K-9 Computer Science Education. In 20th Annual Conference on Innovation and Technology in Computer Science Education. Vilnius: Manuscript in preparation.

Barr, V. & Stephenson, C. (2011). Bringing Computational Thinking to K-12: What is Involved and What is the Role of the Computer Science Education Community ?. doi:10.1145/1929887.1929905

Bloom, B.S. (1956). *Taxonomy of educational objectives: the classification of educational goals*. New York: McKay.

Boersma, K., Eijkelhof, H., van Koten, G., Siersma, D., & van Weert, C. (2006). *De relatie tussen context en concept*. Available from http://www.betanova.nl/downloads/context/relatie__context__concept.pdf

References

Bruner, J.S. (1960). *The Process of Education*. Cambridge, Massachusetts: Harvard University Press.

Bruning, L. & Michels, B. (2013). *Concept- contextvenster*. (Tech. Rep.). Enschede: SLO.

CAS Working Group (2012). *Computer Science: A curriculum for schools*. (Tech. Rep.). Kent: Computing at School.

Cohen, L., Manion, L., & Morrison, K. (2011). Coding and content analysis. In Research Methods in Education. (7 ed., pp. 559–573). New York: Routledge.

CSTA Standards Task Force (2011). *CSTA K-12 Computer Science Standards*. (Tech. Rep.). New York: Computer Science Teachers Association.

De Block & Heene, J. (1987). *Inleiding tot de algemene didactiek*. Antwerpen: Standaard Educatieve Uitgeverij.

Dillon, R.F. (1986). Issues in cognitive psychology and instruction. In Dillon, R.F. & Sternberg, R.J. (Eds.). Cognition and Instrction. Orlando: Academic Press.

Fischer, R. (1984). Unterricht als Prozeßvon der Befreiung vom Gegenstand: Visionen eines neuen Mathematikunterrichts. *Journal fur Mathematik-Didaktik*, *1*, 51–85.

Harden, R.M. (1999). What is a spiral curriculum?. *Medical teacher*, *21*(2), 141–143. doi:10.1080/01421599979752

IEEE (2011). *A Guide to the Project Management Body of Knowledge*. (Tech. Rep., pp. 1–508). Author. doi:10.1109/IEEESTD.2011.6086685

ISO 1087-1 (2000). *Terminology Work—Vocabulary—Part 1: Theory and Application*. (Tech. Rep.). Geneva: International Organization for Standardization. Available from http://www.iso.org/iso/catalogue_detail.htm?csnumber=21197

KNAW (2012). *Digitale geletterdheid in het voortgezet onderwijs*. (Tech. Rep.). Amsterdam: KNAW. Available from http://www.knaw.nl/Content/Internet_KNAW/publicaties/pdf/20121027.pdf

Krathwohl, D.R. (2002). A Revision of Bloom's Taxonomy: An Overview. *Theory Into Practice*, *41*(4), 212–218. doi:10.1207/s15430421tip4104_2

Ministère de l'Éducation nationale (2012). *Enseignement de spécialité d'informatique et sciences du numérique de la série scientifique - classe terminale.*

Available from http://www.education.gouv.fr/pid25535/bulletin_officiel.html?cid_bo=57572

Nievergelt, J. (1990). Computer Science for Teachers: A quest for classics and how to present them. In Norrie, D.-H. & Six, H.-W. (Eds.). Lecture Notes in Computer Science. Berlin, Germany: Springer.

OECD (2013). *PISA 2015 Draft Science Framework*. (Tech. Rep.). OECD.

Ottevanger, W., Oorschot, F., Spek, W., Boerwinkel, D.J., Eijkelhof, H., de Vries, M., ... Kuiper, W. (2014). *Kennisbasis natuurwetenschappen en technologie voor de onderbouw vo*. (Tech. Rep., p. 175). Enschede: SLO.

Oxford University (2010). *Oxford dictionary of English*. (3rd editio ed.). Oxford: Oxford University Press. doi:10.1093/acref/9780199571123.001.0001

Schraw, G. (1998). Promoting general metacognitive awareness. *Instructional Science*, *26*(1), 113–125. doi:10.1023/A:1003044231033

Schreiber, A. (1983). Bemerkungen zur Rolle universeller Ideen im mathematischen Denken. *Mathematica Didactica*, *6*, 65–76.

Schwill, A. (1994). Fundamental ideas of computer science. *European Association for Theoretical Computer Science Buletin*, *53*, 274–295.

_____ (2004). Philosophical Aspects of Fundamental Ideas: Ideas and Concepts. In Magenheim, J. & Schubert, S. (Eds.). Informatics and Student Assessment–Concepts of Empirical Research and the Standardisation of Measurement in the Area of Didactics of Informatics. Bonn: Köllen Druck + Verlag.

SLO (2007). *Examenprogramma informatica havo/vwo*. (Tech. Rep.). Enschede: Stichting Leerplan Ontwikkeling.

Strauss, A.L., Corbin, J.M., & Others (1990). *Basics of qualitative research*. (2nd ed.). Newbury Park, CA: Sage.

Tedre, M. (2014). *The Science of Computing: Shaping a Discipline*. Boca Raton: CRC Press.

The National Research Council (2012). *A Framework for K-12 Science Education*. (Tech. Rep.). Washington: National Academies Press.

Zendler, A. & Spannagel, C. (2008). Empirical Foundation of Central Concepts for Computer Science Education. *Journal on Educational Resources in Computing*, *8*(2), 1–15. doi:10.1145/1362787.1362790