Informatics in Education, 2021, Vol. 20, No. 4, 615–639

DOI: 10.15388/infedu.2021.27

Abstraction in Computer Science Education: An Overview

Claudio MIROLO[1], Cruz IZU[2], Violetta LONATI[3], Emanuele SCAPIN[1]
[1]*Dept. of Mathematics, Computer Science and Physics, University of Udine, Italy*
[2]*The University of Adelaide, Australia*
[3]*Dept. of Computer Science, University of Milan, Italy*
e-mail: claudio.mirolo@uniud.it, cruz.izu@adelaide.edu.au, lonati@di.unimi.it, emanuele.scapin@uniud.it

Received: February 2021

Abstract. When we "think like a computer scientist," we are able to systematically solve problems in different fields, create software applications that support various needs, and design artefacts that model complex systems. Abstraction is a soft skill embedded in all those endeavours, being a main cornerstone of computational thinking. Our overview of abstraction is intended to be not so much systematic as thought provoking, inviting the reader to (re)think abstraction from different – and perhaps unusual – perspectives. After presenting a range of its characterisations, we will explore abstraction from a cognitive point of view. Then we will discuss the role of abstraction in a range of computer science areas, including whether and how abstraction is taught. Although it is impossible to capture the essence of abstraction in one sentence, one section or a single paper, we hope our insights into abstraction may help computer science educators to better understand, model and even dare to teach abstraction skills.

Keywords: computer science education, abstraction, computational thinking, concept development.

> *We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases is called "abstraction".*
>
> Edsger W. Dijkstra (1972b)

1. Introduction

There has always been broad agreement within the education community that abstraction skills play a major role in computer science – see e.g. Dijkstra (1972b); Kramer (2007); Hazzan (2008); Grover and Pea (2018). As observed by Eckerdal *et al.* (2006),

"[i]f one searches the papers available through the ACM Digital Library [...] using the keyword 'abstraction', 63% of all articles are found." In this vein, not surprisingly, Jeannette Wing claimed that "[t]he most important and high-level thought process in computational thinking is the abstraction process", especially in that it "gives us the power to scale and deal with complexity" (Wing, 2011).

The role of abstraction in our field is indeed very pervasive, making abstraction difficult to describe in a nutshell. In fact, abstraction is usually referred to in overly general terms, leading Cetin and Dubinsky to assert that "[a]lthough researchers have accepted that abstraction is a central concept in computational thinking, they are quick to disagree on the meaning of it" (Cetin and Dubinsky, 2017). We can, however, find some consensus, as stated by Hazzan (1999) on the fact that "the notion of abstraction can be examined from various perspectives, that certain types of concepts are more abstract than others, and that the ability 'to abstract' is an important skill."

The struggle to define abstraction is an issue not only for computer scientists; even in mathematics, a field built around abstractions, the perception of this mental ability is far from consolidated, as remarked by Scheiner and Pinto (2016) in a paper discussing "the contradictions, controversies, and convergences concerning the many images of abstraction." A comparison of abstraction's role in computer science and in mathematics is discussed in Verhoeff (2011), from an educational standpoint, and in Colburn and Shute (2007), from a more general philosophical perspective. According to Verhoeff (2011), algorithms and models of data structures are abstractions with a clear mathematical flavour, whereas the abstractions envisaged to model some specific application domain can be put on a par with the abstractions found in other sciences, and notably in physics. On the other hand, Colburn and Shute (2007) argue that the role of abstraction in computer science may be "fundamentally different" from and richer than that implied in mathematics.

Since the literature on abstraction is quite extensive, in order for the overview to be both meaningful and viable we focus its scope on three specific outcomes:

1. To capture different characterisations of abstraction that are relevant to computer science educators at any level, including teachers with limited expertise.
2. To outline examples and contexts for such characterisations in different computer science areas.
3. To review and provide pointers to key papers addressing the cognitive and pedagogical implications of abstraction.

The rest of the paper is organised as follows. Section 2 summarises a range of interesting characterisations of abstraction emerging from different fields. In Section 3 we will take a cognitive point of view and consider the implications for learning; in particular, we discuss an important aspect from an educational perspective: the *abstraction as a process* vs. *abstraction as a product* dichotomy analysed in some depth by White and Mitchelmore (1999). Then, the subject of Section 4 is abstraction in computer science. We will describe how abstraction plays a range of roles in computational thinking, programming and software development, as well as system modelling. In particular, we will focus on the ability to deal with different levels of abstraction simultaneously, a skill of paramount relevance in computer science. In Section 5 we will consider the

issues concerning the teaching and the assessment of abstraction skills. In this respect, the pedagogical practice is affected by the difficulties to operationalise abstraction in order to monitor the learning outcomes, what led to diverging approaches addressing abstraction either directly or indirectly. Finally, Section 6 will present conclusions and future perspectives.

2. Characterisations of Abstraction

Notions of abstraction apply to a broad variety of thought processes. In some way, virtually every useful piece of learning implies some sort of abstraction *from* contingent experiences, and it means different things to different educators. This section will review some features from those various perspectives that are commonly associated with abstraction, exemplifying them with tasks and situations that occur in computer science. We will start from the basic meaning usually assigned to abstraction.

2.1. *Basic Definitions of Abstraction*

From its Latin roots, abstraction should basically mean "something pulled or drawn away." The Oxford English Dictionary describes abstraction in lay terms (definition 3a) as the "action of considering something in the abstract, independently of its associations or attributes; the process of isolating properties or characteristics common to a number of diverse objects, events, etc., without reference to the peculiar properties of particular examples or instances."[1]

White and Mitchelmore (1999) built upon a notion of abstraction as a constructive process with dual actions of "recognizing similarities and ignoring differences." This is reminiscent of Locke's view of abstraction as a two-sided process, where some features are retained, whereas some other features are ignored. This simple characterisation resonates with many computer scientists, for example:

- "Abstraction is the elimination of the irrelevant and the amplification of the essential" (Martin, 2003).
- "The abstraction process – deciding what details we need to highlight and what details we can ignore" (Wing, 2008).
- "Abstraction is defined by having the ability to determine which aspects are important and which are not" (Rijke *et al.*, 2018).
- "Abstraction is the process of simplifying and hiding detail to get at the essence of something of interest" (Curzon *et al.*, 2019).

Extracting similarities. Probably, the most basic meaning of abstraction refers to the process of recognising common features in different examples, and create or define a new category that groups all the objects with such features – including hence the exam-

[1] `https://www.oed.com/view/Entry/766`

ples. Note this view matches the Collins' dictionary entry for abstraction:[2] "the process of formulating generalized ideas or concepts by extracting common qualities from specific examples." In other words, non-essential features will present variations, while core attributes will remain unchanged.

This process, referred to as *empirical abstraction* by Piaget *et al.* (1969), plays a fundamental role in the construction of everyday concepts from the perceptions of superficial similarities between observed objects or facts. In Cetin and Dubinsky (2017) this process, revisited in the computational thinking (CT) context, is called *extraction*.

Abstraction activities aimed at primary/middle school level often target this perspective: for example by exploring similarities in nursery rhymes and looking for algorithmic (in particular, iterative or recursive) patterns to explain their structure (Di Vano and Mirolo, 2011). Similarly, this simple abstraction *process* can support school teachers to introduce the `repeat` construct in Logo or Scratch. After learning to draw lines and experimenting with the turtle commands `forward` and `right` (by 90 degrees), children are assigned the task of drawing a square, usually resulting into coding a sequence of four pairs of commands `forward+right`. Then, the pupils are invited to recognise the recurrent pattern and to appreciate the usefulness of being able to apply a `repeat` (4 times) command. On this basis, the abstraction level could subsequently be raised even more via parameterisation, a form of abstraction as *generalisation*, guiding the learners to draw a variety of regular polygons with different numbers and/or sizes of sides.

This process is also used at higher instructional levels, such as the recognition of common patterns among related problems in the *pattern-oriented* approach of Muller and Haberman (2008). We will return briefly on this in Section 5.1.

Ignoring non-essential features. When extracting similarities from examples, as described above, other specific features are ignored as non-essential. Several philosophical, mathematical, and scientific views of abstraction give prominence to this elimination of non-essential features in connection with some specific objectives. This perspective is particularly important in computer science, as we will discuss in Section 4.

Here we start by just mentioning a couple of illustrative examples. The first one is an unplugged CT abstraction task addressed to primary students,[3] who are asked to read nouns from a pack of cards and quickly draw *sketches* for a partner to guess what they are representing. In doing so they learn that they are ignoring unimportant details and only including that which is most important.

The second one, the concept of *variable*, is among the earliest abstractions students find when learning to program. A variable, together with its associated type, is an abstraction for a memory location that contains a binary representation. Students see variables of different types, use them in simple programs and learn to trace their values, without being concerned with the details of the actual underlying representations – and, as far as possible, without having to care about the possible related limits.

[2] Collins English Dictionary at `https://www.thefreedictionary.com/Abstractions`

[3] `https://www.barefootcomputing.org/resources/abstraction-unplugged-activity`

2.2. *The Generative Power of Abstraction*

The dual operations of extracting common features and ignoring peripheral details lead quite naturally to generalisations, a mental process usually connected to abstraction. Narrowing down the process of abstraction to generalisation may sound appealing at first sight, but at a closer scrutiny it results inadequate, all the more so when abstracting from concepts instead of examples. As a matter of fact, abstraction is also a method to build new, original mental objects, which are not simply the result of bottom-up generalisations. As remarked by Ferrari (2003), "interpreting abstraction as just generalization" overlooks "cognitive and linguistic requirements," where the "linguistic requirements" include trying to attribute meaning to the newly introduced terminology. Thus, in this sense, abstraction should be seen as a powerful, generative and creative process.

This idea can be illustrated by the concept of graph, a data structure that can be used to represent binary relations over a set of objects. Graphs are used in computer science to model a variety of situations (e.g., road maps, dynamical systems, social networks, the World Wide Web, and so on); however, this flexible concept could not just emerge as a bottom-up generalisation from a range of situations.

Embracing Cetin and Dubinsky's suggestion, abstraction can then be thought of as getting to the *essence* of a concept, and "making mental constructions to form that essence," i.e., "a description of the concept that is independent of any context and hence can apply to the concept in all situations in which it appears" (Cetin and Dubinsky, 2017). This perspective has its roots in Piaget's *reflective abstraction*, a higher-level form of abstraction following the more spontaneous empiricalabstraction. In reflective abstraction, knowledge is not drawn from some objects' properties, but mostly from the individual's thinking and mental processes (what Piaget calls the "general coordination of actions").

We will exemplify this point with a standard program comprehension task. To grasp what a (previously unknown) program does – i.e., to figure out the algorithm it implements – a programmer has to build a mental model accounting for the program's behaviour as a whole, which requires the ability to interpret code at different abstraction levels (Izu *et al.*, 2019). This process clearly goes beyond understanding each line of code, or being able to trace the program for specific input data; it typically proceeds through the identification of *beacons* (indicators of particular structures or operations) and *chunks* (meaningful portions of code). Incidentally, this is a task novice programmers often struggle to cope with, lacking the ability to "see the forest for the trees" (Lister *et al.*, 2006).

The generative role of abstraction is prominent in the development of scientific concepts, as pointed out by Chambers (1991). While a large amount of practical knowledge is built upempirically in terms of generalisations, "[s]cientific conceptsand explanations [...] are not defined by reference to observation [and] do not consist of descriptions of observable facts." Rather, "observation in science is defined by reference to the abstract scientific concepts, theories, and explanations." Said otherwise, "abstraction does not proceed by summarizing observations, but by generating a nonobservational structure which deliberately does not summarize" (Judith & David Wilier, 1972, cited in Cham-

bers (1991)). Theoretical ideas such as gravitation and electromagnetic waves in physics, or natural selection in biology were not the outcome of an inductive process resulting from the identification of recurrent patterns (Chambers, 1991). On the contrary, ideas like these usually come before and guide the gathering as well as the interpretation of the observed data (Ohlsson and Lehtinen, 1997).

In short, powerful ideas are usually not the outcome of generalising empirical observations, but require some unprecedented vision of a creative mind. Chambers' point also applies to theoretical models in computer science, such as the "Turing machine" model of computation, the computational complexity classes, or Shannon' theory of information.

3. Abstraction from a Cognitive Point of View

Before we examine the role of abstraction in the computer science field, this section summarises theoretical studies that appraise abstraction from a cognitive point of view.

3.1. *Abstraction as a Product vs. Abstraction as a Process*

The need to bring about suitable conditions in order for the students to be able to experience the *process* of abstraction has been put forward by several researchers on the learning of mathematical concepts. In this respect, for instance, Mason (1989) wrote that "student's sense of abstract as *removed from* or *divorced from* reality (or perhaps, more accurately, from meaning, since our reality consists in that which we find meaningful) [...] arises because there has been little or no participation in the process of abstraction [...]. Despite current emphasis on exploration and investigation, many students may still not experience the shift of abstraction unless they receive explicit assistance."

Later, White and Mitchelmore (1999) dug a little more deeply into the implications of process vs. product approaches when introducing new mathematical concepts. They remarked that most teaching practices introduce abstractions as ready-to-use products, in a (context-free) *abstract-apart* way, rather than focusing on the context-situated processes which could give rise to them and so promoting the learning of what they called *abstract-general* concepts. As a result, in their opinion, "concepts and procedures learnt in an abstract-apart manner are limited because they can only be applied in situations which look suitably similar to the context-free way in which they were learnt."

For the sake of clearness, abstraction as a process is perhaps a little more subtle than it may appear at first. The *process* should not be merely meant as the ability to map something concrete – e.g., a real building – into some related abstraction – say its schematic layout. This can be achieved as well after encountering the "building layout" abstraction, or rather its conventional representation, as a product, once the connections between items in the concrete object and corresponding artefacts in its abstracted representation are understood. Instead, *abstraction as a process* means experiencing abstraction as the result of engaging in a process that leads to realising the need and/or

usefulness of that abstraction in order to focus on what is relevant to achieve the task at hand. And note, in this respect, the decision to leave some details out and include others is based on our aims while abstracting over some object, not on the object itself. Following Wilensky (1991), abstractness "should not be considered as an inherent property of some object, but of the relation of someone's mind to the object." For example, we could provide a range of schematic layouts for the same building; all of them will reflect the same walls and corridor distribution, but depending on our needs or goals, we may want to capture electrical details (charging points, lights), wall treatments, fixtures, etc.

The distinction between experiencing abstractions as either products or processes also applies to the basic forms of *procedural* and *data* abstractions, so pervasive in the programming activity. Typically, novice learners struggle to devise neat solutions based on these forms of abstraction precisely because they were exposed to a number of examples presented as products, made available by someone else more expert – the teacher, the textbook's author – without taking part in a *process* giving rise to the design decisions.

Let us also consider the example of a theoretical abstraction in computer science. The *big-O* notation is an intrinsically abstract concept that is usually introduced at tertiary level to categorise algorithm performances. A characterisation in terms of *big-O* notation is clearly an *abstract* characterisation *as opposed to* the more *concrete* one consisting in a table of running times obtained empirically by running a program on a sample set of inputs. A standard way of presenting the *big-O* notation starts from its formal definition in terms of the mathematical concepts of bounding constants and limits, and then proceeds by explaining how it can be used in the analysis of a few algorithms. With a similar approach, abstraction is introduced as a conceptual *product* we are just trying to apply in sample situations, and the burden of grasping its nature and relevance is left to the learners.

To experience *abstraction* as a *process*, on the other hand, students should be guided through a path starting from what they perceive as concrete at a given learning stage (code, input data, running some code...) and revealing *why* that form of abstraction makes sense in connection with the algorithm properties it is intended to capture. This may be achieved, for instance, by analysing the trends of rates of running times for pairs of programs in the same complexity class vs. pairs of programs belonging in different classes.

3.2. *Structural vs. Operational Dichotomy*

This dichotomy dates back to Piaget's distinction between "figurative" (structural) and "operative" modes of thought (Piaget, 1969). Figurative modes account for representing static aspects of reality, whilst operative modes enable us to make sense of the dynamic and transformational aspects of reality. The operative mode affects how we perceive and build mental images of objects from our experience; conversely, the figurative mode provides objects of thought that can be acted upon and transformed under the operative mode.

Operational and Structural views. In her theoretical framework, Sfard (1991) distinguishes between two different modes of conceiving (abstract) mathematical notions: *operationally* (as processes[4], algorithms, or actions) or *structurally* (as objects). For instance, a circle can be seen operationally as the curve obtained by rotating a compass around a fixed point, or structurally as the locus of points that are equidistant from a given point.

When building mathematical notions (while learning them, or in their historical development), the operational conceptions occur before the structural ones, which Sfard deems more abstract. In her words, seeing an entity "as an object means being capable of referring to it as if it was a real thing [...]. It also means being able to recognize the idea 'at a glance' and to manipulate it as a whole, without going into details. [...] In contrast, interpreting a notion as a process implies regarding it as a potential rather than actual entity, which comes into existence upon request in a sequence of actions" (Sfard, 1991, p. 4).

Operational conceptions are likely to play a significant role in the formation of computing concepts as well, which makes Sfard's model worth considering here. Take, for example, the programming concept of *iteration*: an operational conception of *iteration* relates to actually carrying out the repetition of a sequence of given instructions in concrete situations; a structural conception, instead, sees *iteration* more comprehensively as a programming construct. The latter view is required in order to understand a sentence like "sequencing, selection, and iteration are the building blocks of structured programming."

Similarly, the notion of algorithm, which can be defined operationally as a sequence of steps to be executed to solve a problem, can be conceived structurally as a computation strategy achieving some desired behavior (e.g., in terms of input-output relation). The operational conception is sufficient to trace, i.e., simulate the execution of a given algorithm, whereas one needs to conceive the notion structurally in order to understand its properties, such as correctness, termination or computational complexity.

Concept development process. Sfard dissects the process of concept development into three stages: *interiorization*, *condensation* and *reification*. At the first stage, the learner gets acquainted with applying some procedures; then, at the intermediate stage, they start to see a given process as a whole. Eventually, the reification stage marks an "ontological shift," i.e., a sudden leap to seeing "something familiar in a totally new light." The new entity is now detached from the process that originated it, and it can become a new basic unit for higher level processes.

A similar perspective is taken by Dubinsky (1991) who builds upon Piaget's notion of reflective abstraction to investigate the development of advanced mathematical concepts. His theory – that has also been applied to computational thinking (Cetin and Dubinsky, 2017) – is referred to as APOS, from the acronym of the four implied mental structures. According to Dubinsky's theory, a mathematical notion is first understood as action (A), i.e., a transformation involving physical or mental objects that can

[4] Sfard's term "process" is meant here as a synonym of algorithmic procedure, not to be confused with White and Mitchelmore's use of the term to refer to the action of developing an abstraction.

be carried out by concrete means. After repeated practice on the same action as well as reflection on its implications, the learner may *interiorise* the action and become able to perform it mentally as a process (P). The process may then be *encapsulated* into a mental object (O), as the learner realises that the process can be operated on as a whole. Indeed, what distinguishes an object from a process is precisely this potential to act on it. Whenever necessary, an object can be *de-encapsulated* to go back to the process that originated it. Eventually, a schema (S) emerges when these constructs are organised and linked into a coherent framework, that can then become a new object for higher level schemas.

3.3. *Multiple Representations Perspective*

Duval's analysis of mathematics cognition provides a complementary perspective of abstraction. Although Duval (2006) actually makes an explicit use of the term "abstract" only once in his paper, the reference to the abstract nature of mathematical concepts is implied by asserting that "[m]athematical objects [...] are never accessible by perception or by instruments [...]. The only way to have access to them and deal with them is using signs and semiotic representations" – which are usually multifold. As a consequence, according to Duval, two kinds of transformations play a central role: *treatment* and *conversion*. Treatments are essentially algorithmic (in Sfard's sense) manipulations within a given semiotic register, while conversions connect different kinds of representations and provide a mapping between them. Conversions are cognitively quite complex since they presuppose the recognition of the same denoted object, which must be dissociated from its representation(s). This characterisation resonates with both the view of abstraction as *getting to the essence* – e.g. (Cetin and Dubinsky, 2017) – and Sfard's idea of abstract entities obtained via reification (Sfard, 1991).

In Duval's view, the peculiar thinking processes of mathematics require the cognitive coordination of different semiotic representations in order to compensate for the lack of direct (or instrumental) access to an entity. Since computer science is also abstract in nature, it is conceivable that treatments and conversions play a similar role for the related concepts. In particular, programmers constantly have to do with *conversions* between different types of representations of a same ideal model or entity. For the sake of concreteness, think e.g. of changing the internal structure of a data abstraction while keeping the external "contract" semantics unchanged; or think of moving from flow-charts to textual code (and viceversa) to represent an algorithm.

Mirolo and Di Vano (2013) have drawn from Duval's valuable insights while designing a range of activities to introduce computing ideas in the middle-school. Following their proposal, students are guided to work with heterogeneous representations, including unplugged paper and cardboard artefacts, in order to explore both the (structured) state of the computation and the very idea of algorithmic procedure. Such "explorations" are precisely meant to develop a more abstract view of the implied concepts.

3.4. *Analogies and Metaphors*

Analogies and metaphors can be helpful vehicles for the cognitive assimilation of novel ideas. An inspiring theme arising from science education concerns the exploitation of *analogies* to enable an intuitive grasp and eventually trigger the construction of an abstract concept. When relevant insight is drawn from analogies between a range of sources, the cognitive role such "model" sources play is somehow similar to that of multiple representations. Pedagogical approaches based on similar uses of analogies have been proposed, for instance, in order to mitigate possible misconceptions due to oversimplification of the relationships to the target (Spiro *et al.*, 1989) – and even with partially understood sources (Kurtz *et al.*, 2001). Moreover, as an additional instantiation of this perspective, Podolefsky and Finkelstein (2007) propose the use of multiple, layered analogies "as stepping stones toward more deeply structured abstract reasoning" to scaffold the learning of abstract ideas in physics.

A related concept to analogy is that of *metaphor*. According to Colburn and Shute (2008), "an abstraction is created to manage complexity through information hiding, and a *metaphor* is used to name the abstraction" (whereas the "information hiding" is to be described in non-metaphorical terms). This kind of metaphors "expand the ontological framework of our language for talking about computational processes [and] work their way into programming languages." Colburn and Shute identify three main roles of computer science metaphors: (1) a *pedagogical role*, to facilitate the accommodation and assimilation (in Piaget's terminology) of new abstractions; (2) a *design-oriented role*, in particular when designing a friendly interface, evoking a context familiar to the intended user; and (3) a *scientific role*, to explain computing concepts.

4. Abstraction in Computer Science

In this section we will consider the role of abstraction in typical computer science tasks, including the wider scope of computational thinking (CT). We first review what is meant by abstraction in computational thinking literature, then we illustrate specific abstractions that are used in the field of programming and software development; finally we discuss the importance of being able to move between different levels of abstraction.

4.1. *Abstraction in Computational Thinking*

Since Wing's seminal paper (Wing, 2006) on computational thinking, a large body of work has focused on revising and providing an operational definition of computational thinking, with particular emphasis on K-12 level, in order to help educators to build CT skills across a range of curriculum's areas.

The CT literature recognises the role of abstraction in different contexts and for a variety of purposes; in particular, when modeling real-world problems into computational problems and when developing related artefacts. All the facets of abstraction

discussed in Section 2 – abstraction as generalisation, recognising similarities, ignoring details, and getting to the essence of concepts – play in fact a relevant role in computational thinking.

In Wing (2006) the term abstraction is linked to devising suitable problem representations and mastering complex tasks. Later, she sharpened the emphasis on the role of abstraction by stating that core skills in CT are "defining abstractions, working with multiple layers of abstraction and understanding the relationships among the different layers. Abstractions are the 'mental' tools of computing" (Wing, 2008). In addition, the "operational definition" of CT proposed by ISTE and CSTA includes "representing data through abstractions, such as models and simulations" as one of the key problem-solving processes that characterise CT (Barr *et al.*, 2011).

Barr and Stephenson (2011) underlined the importance of recognising abstractions, of moving between levels of abstraction, of simplifying from the concrete to the general as solutions are designed and developed. They provide examples of how abstraction in CT could be embedded in different disciplines, e.g., when identifying essential facts in a word problem (math) or building a model of a physical entity (physics).

Shute *et al.* (2017) provided a good summary of how CT has been characterised in the literature. Six major aspects of CT emerge from their analysis: decomposition, abstraction, algorithm design, debugging, iteration, and generalisation. In particular, they define abstraction as being able to "extract the essence of a (complex) system," and identify three subcategories (Shute *et al.*, 2017, Table 4):

1. Data collection and analysis: collect the most relevant and important information from multiple sources and understand the relationships among multilayered datasets.
2. Pattern recognition: identify patterns/rules underlying the data/information structure.
3. Modeling: build models or simulations to represent how a system operates, and/or how a system will function in the future.

This is probably the more encompassing definition of abstraction in CT. In other operational definitions or frameworks data structures and/or pattern recognition are not explicitly linked or captured inside *abstraction*. For example, ISTE[5] introduces abstraction and pattern recognition as separate pillars of CT, together with decomposition and algorithms.

As reported in Kite *et al.* (2021), programming (e.g. block-based coding platforms, toy robotics, game development) is usually the main vehicle for teaching CT. This instructional approach is sometimes referred to as *plugged* due to its associated hardware resources.

Plugged CT. Many operational definitions of abstraction in CT are linked to programming. For instance, abstraction is paired up with modularisation in the framework proposed by Brennan and Resnick (2012), which is derived from their studies on children's activities with Scratch.[6] In particular, they characterise the computational practice of

[5] `https://www.iste.org/areas-of-focus/computational-thinking-in-the-classroom`

[6] A widespread visual programming environment that the authors describe as a "computational authoring environment."

"abstraction and modularisation" in terms of "building something large by putting together collections of smaller parts." This practice is observed both in the initial work of conceptualising the problem and in its implementation, structured into individual parts.

Abstraction provides the last letter for the acronym VELA, the label of a project aimed at enhancing the understanding of programming concepts in K-12 (Grover *et al.*, 2019) – the other letters standing for Variables, Expressions, and Looping. In the context of this project, abstraction is defined as "the process of giving a name to a specific collection of details as a way of referencing its purpose without quoting or enumerating its detail." More generally, in their learning goals, abstraction is associated with the identification of patterns, the use of variables to represent data, and the use of multiple representations.

Unplugged CT. Computational thinking provides a multifaceted perspective to approach complex and open-ended problems, which can also be applied to everyday activities, either plugged or unplugged.

The tasks in the Bebras Challenge[7] are popular sources for unplugged CT activities. An ITICSE Working Group in 2015 analysed the scope of Bebras tasks in relation to CT concepts; their operational definition of abstraction for such tasks covered: (1) problems that ask for the creation of a formula, (2) the distillation of broader ideas out of narrower concepts, (3) finding rules that apply to a given problem, (4) finding a pattern to model some behavior, and (5) identifying essential facts about a structure or problem to verify correct answers (Barendsen *et al.*, 2015, Table 4).

Additionally, abstraction plays a central role for a variety of unplugged CT activities in different knowledge fields. In Peel *et al.* (2019), for example, secondary honors biology students learned CT principles and engaged in the design of unplugged algorithmic explanations to describe natural selection for three organisms. After revising each algorithm, they were able to generalise across contexts and produce a generic natural selection algorithm. This is an interesting example of unplugged computational modelling.

Computational abstractions. Independently of working in a plugged or unplugged context, the creation of computational abstractions can be seen as a computational problem-solving practice. As remarked by Weintrop *et al.* (2016), "creating an abstraction requires the ability to conceptualize and then represent an idea or a process in more general terms" by focusing on the important aspects and leaving less relevant features on the background.

A key aspect differentiates CT from other disciplines using abstraction: the "essence of Computational Thinking lies in the creation of 'logical artifacts' that externalize and reify human ideas in a form that can be interpreted and 'run' on computers" (Hoppe and Werneburg, 2019). Said otherwise, the abstractions devised to model real-world problems materialise and become *real* in programs, in that they will live on whenever the program is running – typically with concrete effects on the real lives of their users – without the mediation of minds, which instead is usually required for abstract ideas to exist.

[7] `https://www.bebras.org/` – Annual online competition, started in Lithuania in 2004 and spread all over the world, whose aim is to encourage school students' interest in Informatics and CT

4.2. *Specific Abstractions in Programming and Software Development*

When creating programs, and computational artefacts in general, programmers can refer to a repertoire of computational abstractions and use them as constructive tools, or "concrete abstractions" (as dubbed in Hailperin *et al.* (1999)). In this section we summarise the most common and relevant of such concrete abstractions.

Programming languages and paradigms. Programming languages provide different kinds of abstractions, mostly according to the application domains they are designed for, and can be classified for their level of abstraction. Low-level languages, as assembly, are *close* to machine languages (whose instructions are expressed only by binary digits and are understood directly by the machines) and provide little or no abstraction from the details about how the machine works. High-level languages instead are designed to be easily understood by humans, and indeed they require to be translated into executable programs in order to be run by machines.

Similarly, programming paradigms resonate the computational models they are based upon and can be analyzed according to the abstractions they provide. Indeed the expressive power of all programming languages[8] and paradigms is the same, what differentiates them is the easiness of accessing, using, or building different abstractions. For example, *structured programming* allows to organise the data flows using three basics constructs (sequence, selection, and iteration) and to structure programs with separate modules as functions or procedures, while *object-oriented programming* supports further abstractions as encapsulation, polymorphism, inheritance.

Generalisation and parameterisation. Generalising and decontextualising play a meaningful role in writing programs, so that they can work on a range of different inputs and contexts. Variables are one of the first tools novices learn to use in order to represent a general category of possible values; the complexity of considering many cases is reduced by replacing multiple similar specific entities with a single abstract one.

Additionally, a program unit may be generalised in order to extend its application to different situations; the identity of data is abstracted away by replacing it by parameters with the aim of generalising functions or modules so that they can be applied in different contexts. This scope extension called *abstraction by parameterisation* by Liskov and Guttag (2001); Liskov *et al.* (1977).

Procedural and data abstraction. When calling functions in programs we are interested in *what* the function does and not in the details on *how* it is implemented; this is what is meant by the expression "procedural" (or functional) abstraction. The user of the function does not have access to the details about the algorithms that accomplishes the function: those details are ignored as non-essential, and abstracted away; instead the user can trust the accuracy of the results and has only to know the signature of the function (a convention that describes how to call it).

A similar separation of concerns is relevant for data as well as program modules. Data abstraction refers to the separation between how data can be accessed, used, and

[8] provided that they are Turing complete.

changed and how data is represented. Abstract data types are designed according to this principle: the external interface encapsulates the operations that can be performed on the data, and the implementation defines how data are handled internally.

Procedural and data abstraction allow to reuse modules in different contexts and to change their implementation without affecting the calling module. In the design-by-contract terminology this idea is referred to as *abstraction by specification* (Liskov and Guttag, 2001; Liskov *et al.*, 1977). In order to address the primary need of managing complexity in large programs, the implementation details are abstracted away by decoupling the program units. Procedural and data abstraction occur together in the object-oriented paradigm, where both data and its associated operations are abstracted into objects.

Information hiding and abstraction layers. In procedural and data abstraction, the user is unaware of the details about the function implementation or the internal representation of data. However, in contrast to other disciplines, this information is just *hidden*, rather than ignored, since it must be preserved and used at other levels of the same program (Colburn and Shute, 2007). The concept of *notional machine* (Du Boulay, 1986) is representative of this idea of hidden abstraction. Notional machines are indeed abstract models that provide a context for understanding the program behavior without the need to be aware of the machinery of several layers of hardware and software. Programmers do much (if not all) of their work at the notional machine abstraction level.

This information hiding concept is pervasive in computer science. In complex systems, such as an operating system or a network server, functionalities are broken into multiple layers, so that each layer implements some functionality or service that is provided to the layers above. The higher layers do not need to know any of the details of the implementations in the lower layers, which are hidden to them. They only need to know how to ask for that service. However, although details are hidden, they are relied upon, and dealt with in different layers.

4.3. *Moving through Different Abstraction Levels*

Many scholars have highlighted the importance, in computer science, of being able to move over different levels of abstraction:

- "[T]he arrangement of various layers, corresponding to different levels of abstraction, is an attractive vehicle for program composition" (Dijkstra, 1972a);
- "[T]he essence of computer science is an ability to understand many levels of abstraction simultaneously" (Knuth, 2003);
- "Thinking like a computer scientist [...] requires thinking at multiple levels of abstraction" (Wing, 2006);
- "The abstractions of computer science [...] are constantly changing, requiring multiple, multilayered abstractions of interaction patterns" (Colburn and Shute, 2007).

The latter regarded this ability as very peculiar to the field, and this is particularly manifest in algorithm and program design.

Table 1
PGK Hierarchy levels from Perrenet *et al.* (2005) and their mapping to K-5 settings according to Waite *et al.* (2018).

PGK Level	Definition	K-5 name	K-5 question
Problem	Algorithm perceived as a problem solving strategy	problem	"What is needed"
Object (algorithm)	Algorithm understood independently of any specific implementation	design	"What it should do"
Program	Algorithmic grasp of the program	code	"How it is done"
Execution	Focus on individual runs with specific inputs	running the code	"What it does"

Building on Sfard's work, a first distinction is done by Haberman *et al.* (2005) who argue that "an algorithm can be viewed as an operational process entity (embodying a 'how' view), as well as an object entity that embodies an input/output relationship (and a 'what' view)." Perrenet *et al.* (2005) deepen this approach by defining four abstract levels for the concept of algorithm; the resulting hierarchy (later known in the literature as *PGK hierarchy*) is shown in Table 1. Waite *et al.* (2018) situated this hierarchy in K-5 settings by renaming the levels and associating to each of them the question it pertains.

It is also worth mentioning here Hazzan's "reducing abstraction" framework (Hazzan, 1999). It was proposed at first in relation with the learning of algebra concepts, but subsequently it has also been applied to computer science (Hazzan, 2003; Sakhnini and Hazzan, 2008). Within this framework, a learner's abstraction process is explained in connection with students' tendency to reason at lower levels of abstraction than expected while trying to cope cognitively with new concepts they are learning. Hazzan (1999) argues that the "reducing abstraction" framework is consistent with the most common perspectives on abstraction levels, namely: in terms of the quality of the relationships between the learner and the object of thought (Wilensky, 1991), in terms of the transition from operational to structural abstractions (Dubinsky, 1991; Sfard, 1991), and in terms of the degree of complexity of the concept.

A simple, concrete example of swapping between abstraction levels can occur when completing a task to draw an image on the computer screen. At the design stage, numbers and arithmetic operations are thought of at an abstract level, mathematically. However, when debugging the program code we may need to lower the abstraction to their representation properties in order to make sense of oddities in the visualised drawings that could be explained, e.g., by arithmetic overflow while computing pixel coordinates.

5. Abstraction in the Computer Science Classroom

Although we have presented relevant characterisations of abstraction, it is still far from clear how an abstraction-oriented perspective could become part of the pedagogical practice. Already in the late 1990s, as a revision of the spreading *object-first* orienta-

tion, Machanick (1998) endorsed an *abstraction-first* instructional approach where the implementation of abstract data types is delayed as much as possible in order to stress an abstract view of the models.

Kramer remarked that abstraction per se is not the subject of any computing course, but that all computing courses "rely on or utilize abstraction to explain, model, specify, reason or solve problems," so confirming that "abstraction is an essential aspect of computing, but that it must be taught indirectly through other topics" (Kramer, 2007, p. 41). In line with Kramer's remark, Hazzan (2008) discussed abstraction as a *soft idea*, "that can be neither rigidly nor formally defined, nor is it possible to guide students as to its precise application." And although "it is not a trivial matter," like other soft ideas, abstraction should be taught in a computer science curriculum. Then, a small number of educators have provided guidelines to teach abstraction at different instruction levels. Hence, this section briefly explores their approaches to foster and assess abstraction skills.

5.1. *Teaching to Trigger Abstraction in Computer Science*

Often instructors aim to develop students' abstraction skills indirectly, by devising particular learning trajectories that are supposed to foster higher-level thinking and require students to use abstraction to succeed. In a program development project, for example, they could assign refactoring tasks in which learners are asked to look for recurrent patterns of code and to re-organise the code by introducing meaningful procedural and/or data abstractions with the purpose of making the whole program easier to read, debug and modify. In the following paragraphs we will outline a selection of representative approaches to (an implicit) abstraction-oriented instruction.

Pattern-oriented instruction. This approach has the aim of improving students' competencies in algorithmic problem solving (Muller and Haberman, 2008). An algorithm is indeed seen by these authors as a combination of plan patterns in Soloway's sense (Soloway, 1986), resulting via sequencing, nesting or merging plans from a repertory of basic algorithmic patterns specifically designed for pedagogical purposes.

In Muller and Haberman's scenario, abstraction plays a crucial role in pattern recognition, chunking, and problem structure identification. Their approach relies on having an appropriate pattern repository, as well as on presenting carefully selected problems of gradually increasing difficulty; teachers should then discuss and compare different solutions to a given problem in terms of pattern composition. Additional guidelines for pattern-oriented instruction include: (1) patterns should be abstracted from related examples or by generalising a simpler problem, (2) patterns should be revisited in different contexts, in order to make the identification of common problem features easier, and (3) similarities, differences, and also possible misuses of patterns should be considered. According to Muller and Haberman, comparative studies appear to show that novices exposed to this approach develop enhanced problem solving abilities.

Multiple representations perspective. Dealing with multiple representations of a given phenomenon can play a key role in the development of abstract concepts. Ac-

cording to Ainsworth (2008), in particular, in order "to construct a deeper understanding of a domain," if the learners "fail to relate representations, then processes like abstraction cannot occur. Moreover, although learners find it difficult to relate different forms of representations, if the representations are too similar, then abstraction is also unlikely to occur." She then recommends that teachers should foster abstraction over multiple representations "by providing focused help and support on how to relate representations and giving learners sufficient time to master this process."

In this respect, Gautam *et al.* (2020) have recently proposed an interesting interdisciplinary approach to integrate science (namely, chemistry) and computational thinking in the curriculum. While abstraction is usually "presented as hierarchical" in terms of (i) extracting important features and ignoring unimportant ones, and (ii) finding commonalities across contexts, in their standpoint "abstraction in science" as well as in computing "requires students to move laterally across different representations of the concepts or actions." In the reported study, the micro-level process of photosynthesis was modeled by a code snippet, and by discussing commonalities and differences between, e.g., a whiteboard and the code representation of the implied chemical reaction, "the instructor pushed students towards higher-level abstract thinking." Moreover, they suggest to allow for *friction* emerging when the students explore different representations, in that it encourages to consider alternative views and "negotiate the elements with one another." According to the authors, this approach "created meaningful accounts of science phenomenon and the science provided access to how computation embeds ideas."

Exploration of artefacts. A more recent pedagogical trend in programming education attempts to trigger abstraction through activities inspired by the *use-modify-create* framework. The idea is that the understanding of artefacts such as programs would gradually progress through three major stages, corresponding to (i) exploration via passive use (as a consumer), (ii) experimentation of the internal machinery by modifying some features, and finally (iii) creation of new, original artefacts to achieve specific goals. While discussing the use-modify-create approach, Lee *et al.* (2014) observe that abstraction, as well as other computational thinking abilities, are "not explicitly taught but rather [develop] through one's impetus to create;" nevertheless, in this progression the abilities to modify and, later, to create imply the enhancement of learner's abstraction skills.

5.2. *Teaching the Role of Abstraction in Computer Science*

In contrast to the previous approaches, fewer educators have proposed ways to make abstraction more explicit. Koppelman and van Dijk (2010), for example, reported on students struggling to use procedural abstraction and recommended to teach abstraction early and consciously. What they meant is that instructors should point out where abstraction is used and call it by its name. They should also stress its benefits by comparing solutions with and without procedural abstraction and seeing the former are easier to understand. Nicholson *et al.* (2009) attempted to lay "the basis of a *curriculum of*

abstraction" by delineating fourteen sub-skills involved in working with abstractions. In order to engage students' in using abstraction, they proposed the creation of 'scenarios' that facilitate experimenting with abstractions and allow to get prompt feedback on them. They emphasised the importance of a critical examination of potential uses as well as *mis*uses of programming abstractions in connection with the intended purpose and the different target contexts. Both studies suggested guidelines but did not test them in the classroom. Two explicit approaches that have been put into practice are briefly presented next. For more insights on these contributions the reader is however referred to the cited papers.

Abstraction awareness. Böttcher *et al.* (2016) introduce abstraction explicitly with the goal of fostering "awareness of what abstraction is, and why it is a necessary skill," while making "the cognitive process of abstraction transparent" to the learners. Their intervention consisted on three stages: (1) Testing sessions using abstract tasks, "to make students aware of their lack in abstract thinking," (2) Practice sessions finding commonalities and categorising items from everyday's life, to develop their competence, and (3) a 90 minute lecture that modelled in a collaborative manner the abstract-to-concrete thinking while developing an algorithm, to "deepen the conscious competence" of abstraction.

Abstraction in introductory programming. Armoni and colleagues have developed and explored the potentials of a more comprehensive framework to teach abstraction in computer science explicitly to novices at secondary and tertiary level (Armoni, 2013; Statter and Armoni, 2016). Their approach is based on the PGK hierarchy proposed by Perrenet *et al.* (2005), consisting of four abstraction levels presented in Table 1. Armoni (2013) pointed out that students should be made aware of any change of abstraction level. More specifically, when solving a problem students should become able to appreciate the differences between abstraction levels, to consciously and freely move between different levels, and to decide the appropriate level to achieve a given (sub)task. To pursue these learning objectives, which pertain to the *abstraction-as-process* perspective, Armoni and colleagues suggest a few instructional guidelines, in particular:

- *Proceed from higher to lower abstraction levels*, working at the lower levels only when concreteness is required.
- *Use language cues* as an aid to recognise the intended level (e.g., restrict references to specific programming constructs to the two lower levels).
- *Be persistent and precise* about consistently and clearly distinguishing among levels.
- *Be explicit and reflective*, encouraging students to look back at their own processes.

5.3. *Assessing Abstraction Skills in Computer Science*

"Are abstraction skills assessable at all?" This question has been raised by Hazzan and Kramer (2016), who have asked a group of experts in computer science and software engineering to rate the appropriateness of a range of patterns intended to test students'

abstraction ability. What emerged from their investigation is a 'surprising' disagreement among experts as to the suitability of each proposed type of test, partially compensated by some agreement on requiring students' to *create* themselves abstractions, rather than simply analyse system representations in terms of abstraction. At the core of this question lies, in fact, the need to operationalise the assessment of abstraction skills, an issue that has not been extensively investigated; we will briefly present a range of attempts.

Abstraction in learning taxonomies. We should start by observing that the learning taxonomies of most widespread use in computing education define a scale of increasing abstraction levels. Such a connection is made explicit in the very name "extended abstract" of the topmost SOLO[9] category (Biggs and Collis, 1982), but it is also implied by the *formal generalisation* stage of the Piaget-inspired framework, applied in (Bennedsen and Caspersen, 2006), or by the higher *analysis*, *synthesis* and *evaluation* degrees of Bloom's taxonomy (Anderson *et al.*, 2001).

However, we should be mindful that similar instruments measure general thinking capabilities, not specifically tailored for the computing field. An interesting review of learning taxonomies, leading to a specific adaptation for computer science, can be found in Fuller *et al.* (2007). Moreover, while pointing out that a major source of difficulty to assess abstraction, especially in computer science, can be ascribed to its "inherently context bound nature," Philpott *et al.* (2009) distinguish between 'quantitative' (*multistructural* or below) vs. 'qualitative' (*relational*) SOLO categories, to be interpreted in terms of Sfard's process/object duality (Sfard, 1991), and suggest to use this distinction to design reliable tools to measure abstraction levels in computer science education.

Abstraction in cognitive development. To investigate the impact of abstraction abilities on performance in computer science tasks, Bennedsen and Caspersen (2008) operationalised abstraction in terms of stages of cognitive development, as elaborated by Adey and Shayer (1994), and used a validated test, namely, an adaptation by Adey and Shayer of Piaget's pendulum-test. More specifically, that model identifies eight cognitive development stages: pre-operational, early concrete, mid concrete, late concrete, concrete generalisation, early formal, mature formal, formal generalisation.

Hill *et al.* (2008) devised a three-scale instrument to assess abstraction skills in terms of *conceptual*, *formal* and *descriptive* abstraction. Conceptual abstraction amounts to being able to "see the forest" (Lister *et al.*, 2006), i.e. to being oriented toward the big picture when solving a problem. Formal abstraction refers to the ability to reason with the aid of symbols, i.e. to master the relationships between a formal system and the problem domain. Descriptive abstraction refers to the ability to analyse and identify core parts, similarities and differences between problems.

Abstraction in novice programmers. Statter and Armoni (2020) have reviewed a number of methods used by researchers in the attempt to measure the abstraction level exhibited by students while trying to achieve computing tasks.[10] Their purpose was to devise an appropriate assessment instrument for a study they were planning to carry out,

[9] SOLO is an acronym standing for Structure of Observable Learning Outcome.

[10] We refer the interested reader to the review section in Statter and Armoni (2020) for a broader view on the assessment of abstraction.

and eventually they decided to focus "on a few characteristics of the solution process, which indicate the use of abstraction" (Statter and Armoni, 2016).

A range of *operational* features can be directly related to the already mentioned PGK hierarchy (see again Table 1) such as distinguishing between abstraction levels, recognising the level currently in use or moving freely and smoothly between abstraction levels. Additionally, some "corroborative," indirect evidence of abstraction abilities may come from other complementary indicators such as offering explanations that justify a solution or identifying initialisation (setup) units within programs.

Abstraction skills and transfer. It is worthwhile to conclude this part with a couple of notes about *learning transfer*. Far transfer is indeed a clear indication of abstract thinking, and evidence of its occurrence could then be used to assess the achievement of abstraction skills. In Salomon and Perkins' words, "[h]igh-road transfer occurs by *intentional mindful abstraction* of something from one context and application in a new context" (Salomon and Perkins, 1989)[11]. This task, however, is not a straightforward one either. In spite of recurring optimistic claims that learning to program has the potential to enhance the learners' general problem-solving skills, it is unclear to what extent the abilities acquired through programming can actually transfer to other domains (Pea and Kurland, 1984; Salomon and Perkins, 1989). And according to Guzdial (2015), "there has not been a study since Wing's 2006 paper that has successfully demonstrated that students in a computer science class transferred knowledge from that class into their daily lives."

Ginat *et al.* (2011), on the other hand, have analysed learning transfer among different tasks, all *within* the field of computer science. To this aim, they have characterised students' difficulties in terms of five transfer aspects, namely: *recognition*, *abstraction*, *mapping*, *embedment*, and *flexibility*. There, abstraction is meant as the ability to envisage a general problem-solving pattern, by establishing connections from an analogue (recognition) to the current task (mapping), the latter aspect involving flexibility, to adapt the source pattern, and possibly embedment, to combine more elements to each other. Then, the way they measured transfer is by examining students' solutions to specifically designed tasks, whose features are described in detail in the cited paper.

6. Conclusions

> *I will not solve the problems posed by abstraction, and certainly not the problem of teaching abstraction. But I would like to put it more prominently on the agenda.*
>
> Tom Verhoeff (2011)

Defining abstraction in computer science is challenging as it permeates a variety of tasks including programming, analysing data, creating applications, building large computational models. Faced with such an herculean task, we could not do justice to the many studies that have tackled each abstraction perspective in depth, but we hope this over-

[11] Italic was added by the authors.

view will provide pointers to the interested readers to further investigate the process of abstraction in their own areas of interest.

We have covered abstraction from diverse angles: its basic definition provides an entry point for primary students to grasp computational thinking; the awareness of the role of multiple abstraction levels will help secondary and tertiary students when learning to program; its potential to tackle complex systems, one level at a time, will support software engineers and computational scientists to model real-time problems. We have also seen that abstraction, generalisation and pattern recognition are often twined together in many activities. Moreover, whenever possible we have provided examples of concrete experiences that can be offered as learning opportunities for the students.

In conclusion, we hope this overview will encourage the readers, researchers or educators at any instructional level, to make further progress in the teaching of – and for – abstraction. This can be achieved by introducing and dealing with abstraction more explicitly, emphasising regularly its role, power and benefits, and providing formal or informal feedback on students' progress towards mastering abstraction.

References

Adey, P., Shayer, D.M. (1994). *Really Raising Standards. Cognitive Intervention and Academic Achievement*. Routledge, London, UK. 9780415101455.

Ainsworth, S. (2008). The Educational Value of Multiple-Representations when Learning Complex Scientific Concepts. In: Gilbert, J.K., Reiner, M., Nakhleh, M. (Eds.), *Visualization: Theory and Practice in Science Education*. Springer Netherlands, Dordrecht, pp. 191–208. 978-1-4020-5267-5. https://doi.org/10.1007/978-1-4020-5267-5_9

Anderson, L.W., Krathwohl, D.R., Airasian, P.W., Cruikshank, K.A., Mayer, R., Pintrich, P.R., Raths, D.J., Wittrock, M.C. (2001). *A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. ISBN: 080131903X.

Armoni, M. (2013). On teaching abstraction in computer science to novices. *Journal of Computers in Mathematics and Science Teaching*, 32(3), 265–284.

Barendsen, E., Mannila, L., Demo, B., Grgurina, N., Izu, C., Mirolo, C., Sentance, S., Settle, A., Stupurienundefined, G. (2015). Concepts in K-9 Computer Science Education. In: *Proceedings of the 2015 ITiCSE on Working Group Reports*. ITICSE-WGR '15. Association for Computing Machinery, New York, NY, USA, pp. 85–116. 9781450341462. https://doi.org/10.1145/2858796.2858800

Barr, D.C., Harrison, J., Conery, L. (2011). Computational Thinking: A Digital Age Skill for Everyone. *Learning and Leading with Technology*, 38, 20–23.

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

Bennedsen, J., Caspersen, M.E. (2006). Abstraction Ability as an Indicator of Success for Learning Object-Oriented Programming? *SIGCSE Bull.*, 38(2), 39–43. https://doi.org/10.1145/1138403.1138430

Bennedsen, J., Caspersen, M.E. (2008). Abstraction ability as an indicator of success for learning computing science? In: *ICER '08: Proceeding of the Fourth International Workshop on Computing Education Research*. ACM, New York, NY, USA, pp. 15–26. 978-1-60558-216-0. https://doi.org/10.1145/1404520.1404523

Biggs, J., Collis, K.F. (1982). *Evaluating the Quality of Learning: the SOLO Taxonomy*. Academic Press, New York, USA. 978-0-12-097552-5. https://doi.org/10.1016/C2013-0-10375-3

Böttcher, A., Schlierkamp, K., Thurner, V., Zehetmeier, D. (2016). Teaching Abstraction. In: *Proceedings of the 2nd International Conference on Higher Education Advances, HEAd '16*. Editorial Universitat Politècnica de València, València, Spain, pp. 357–364. https://doi.org/10.4995/HEAd16.2016.2770

Brennan, K., Resnick, M. (2012). New Frameworks for Studying and Assessing the Development of Computational Thinking. In: *Proceedings of the 2012 Annual Meeting of the American Educational Research*

Association. AERA 2012.
Cetin, I., Dubinsky, E. (2017). Reflective abstraction in computational thinking. *The Journal of Mathematical Behavior*, 47, 70–80. https://doi.org/10.1016/j.jmathb.2017.06.004
Chambers, J.H. (1991). The Difference Between the Abstract Concepts of Science and the General Concepts of Empirical Educational Research. *The Journal of Educational Thought (JET)*, 25(1), 41–49. https://doi.org/10.2307/23767703
Colburn, T., Shute, G. (2007). Abstraction in Computer Science. *Minds Mach.*, 17(2), 169–184. https://doi.org/10.1007/s11023-007-9061-7
Colburn, T.R., Shute, G.M. (2008). Metaphor in computer science. *Journal of Applied Logic*, 6(4), 526–533. https://doi.org/10.1016/j.jal.2008.09.005
Curzon, P., Bell, T., Waite, J., Dorling, M. (2019). Computational Thinking. In: Fincher, S.A., Robins, A.V. (Eds.), *The Cambridge Handbook of Computing Education Research*. Cambridge Handbooks in Psychology. Cambridge University Press, Cambridge, pp. 513–546. Chap. 17. https://doi.org/10.1017/9781108654555.018
Di Vano, D., Mirolo, C. (2011). "Computer Science and Nursery Rhymes": A Learning Path for the Middle School. In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ITiCSE '11. ACM, New York, NY, USA, pp. 238–242. 978-1-4503-0697-3. https://doi.org/10.1145/1999747.1999815
Dijkstra, E.W. (1972a). Notes on Structured Programming. In: Hoare, C.A.R. (Ed.), *Structured Programming*. Academic Press Ltd., London, UK. 0122005503.
Dijkstra, E.W. (1972b). The Humble Programmer. *Commun. ACM*, 15(10), 859–866. https://doi.org/10.1145/355604.361591
Du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2, 57–73.
Dubinsky, E. (1991). Reflective Abstraction in Advanced Mathematical Thinking. In: Tall, D. (Ed.), *Advanced Mathematical Thinking*. Springer Netherlands, Dordrecht, pp. 95–126. 978-0-306-47203-9. https://doi.org/10.1007/0-306-47203-1_7
Duval, R. (2006). A Cognitive Analysis of Problems of Comprehension in a Learning of Mathematics. *Educational Studies in Mathematics*, 61(1–2), 103–131. https://doi.org/10.1007/s10649-006-0400-z
Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M., Sanders, K., Zander, C. (2006). Putting threshold concepts into context in computer science education. In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITICSE '06. ACM, New York, NY, USA, pp. 103–107. 1-59593-055-8. https://doi.org/10.1145/1140124.1140154
Ferrari, P.L. (2003). Abstraction in Mathematics. *Philosophical Transactions: Biological Sciences*, 358(1435), 1225–1230. http://www.jstor.org/stable/3558214
Fuller, U., Johnson, C.G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T.L., Thompson, D.M., Riedesel, C., Thompson, E. (2007). Developing a Computer Science-specific Learning Taxonomy. In: *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '07. ACM, New York, NY, USA, pp. 152–170. https://doi.org/10.1145/1345443.1345438
Gautam, A., Bortz, W., Tatar, D. (2020). Abstraction Through Multiple Representations in an Integrated Computational Thinking Environment. In: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. SIGCSE '20. Association for Computing Machinery, New York, NY, USA, pp. 393–399. 9781450367936. https://doi.org/10.1145/3328778.3366892
Ginat, D., Shifroni, E., Menashe, E. (2011). Transfer, Cognitive Load, and Program Design Difficulties. In: Kalaš, I., Mittermeir, R.T. (Eds.), *Informatics in Schools. Contributing to 21st Century Education*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 165–176. 978-3-642-24722-4. https://doi.org/10.1007/978-3-642-24722-4_15
Grover, S., Pea, R. (2018). Computational Thinking: A Competency Whose Time Has Come. In: Sentance, S., Barendsen, E., Carsten, S. (Eds.), *Computer Science Education: Perspectives on Teaching and Learning in School*. Bloomsbury Academic, London, UK, pp. 19–38.
Grover, S., Jackiw, N., Lundh, P. (2019). Concepts before coding: non-programming interactives to advance learning of introductory programming concepts in middle school. *Computer Science Education*, 29(2-3), 106–135. https://doi.org/10.1080/08993408.2019.1568955
Guzdial, M. (2015). *Learner-Centered Design of Computing Education: Research on Computing for Everyone*. Synthesis Lectures on Human-Centered Informatics. Morgan & Claypool Publishers, Williston, USA. https://doi.org/10.2200/S00684ED1V01Y201511HCI033
Haberman, B., Averbuch, H., Ginat, D. (2005). Is It Really an Algorithm: The Need for Explicit Discourse.

In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '05. Association for Computing Machinery, New York, NY, USA, pp. 74–78. 1595930248. https://doi.org/10.1145/1067445.1067469

Hailperin, M., Kaiser, B., Knight, K. (1999). *Concrete Abstractions*. Brooks/Cole, Pacific Grove, CA. http://www.gustavus.edu/+max/concrete-abstractions.html

Hazzan, O. (1999). Reducing Abstraction Level When Learning Abstract Algebra Concepts. *Educational Studies in Mathematics*, 40(1), 71–90. http://www.jstor.org/stable/3483306

Hazzan, O. (2003). How Students Attempt to Reduce Abstraction in the Learning of Mathematics and in the Learning of Computer Science. *Computer Science Education*, 13(2), 95–122. https://doi.org/10.1076/csed.13.2.95.14202

Hazzan, O. (2008). Reflections on Teaching Abstraction and Other Soft Ideas. *SIGCSE Bull.*, 40(2), 40–43. https://doi.org/10.1145/1383602.1383631

Hazzan, O., Kramer, J. (2016). Assessing Abstraction Skills. *Commun. ACM*, 59(12), 43–45. https://doi.org/10.1145/2926712

Hill, J.H., Houle, B.J., Merritt, S.M., Stix, A. (2008). Applying Abstraction to Master Complexity. In: *Proceedings of the 2nd International Workshop on The Role of Abstraction in Software Engineering*. ROA '08. Association for Computing Machinery, New York, NY, USA, pp. 15–21. 9781605580289. https://doi.org/10.1145/1370164.1370169

Hoppe, H., Werneburg, S. (2019). *Computational Thinking – More Than a Variant of Scientific Inquiry!*, pp. 13–30. + supplement. 978-981-13-6528-7. https://doi.org/10.1007/978-981-13-6528-7_2

Izu, C., Schulte, C., Aggarwal, A., Cutts, Q., Duran, R., Gutica, M., Heinemann, B., Kraemer, E., Lonati, V., Mirolo, C., *et al.* (2019). Fostering Program Comprehension in Novice Programmers -Learning Activities and Learning Trajectories. In: *Proc. of the Working Group Reports on Innovation and Technology in Computer Science Education*. ITiCSE-WGR '19. ACM, New York, NY, USA, pp. 27–52. 9781450368957. https://doi.org/10.1145/3344429.3372501

Kite, V., Park, S., Wiebe, E. (2021). The Code-Centric Nature of Computational Thinking Education: A Review of Trends and Issues in Computational Thinking Education Research. *SAGE Open*, 11(2), 21582440211016418. https://doi.org/10.1177/21582440211016418

Knuth, D.E. (2003). Bottom-up Education. In: *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '03. Association for Computing Machinery, New York, NY, USA, p. 2. 1581136722. https://doi.org/10.1145/961511.961514

Koppelman, H., van Dijk, B. (2010). Teaching Abstraction in Introductory Courses. In: *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*. ITiCSE '10. Association for Computing Machinery, New York, NY, USA, pp. 174–178. 9781605588209. https://doi.org/10.1145/1822090.1822140

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

Kurtz, K., Miao, C., Gentner, D. (2001). Learning by Analogical Bootstrapping. *Journal of the Learning Sciences*, 10, 417–446.

Lee, I., Martin, F., Apone, K. (2014). Integrating Computational Thinking Across the K–8 Curriculum. *ACM Inroads*, 5(4), 64–71. https://doi.org/10.1145/2684721.2684736

Liskov, B., Guttag, J. (2001). *Program Development in Java*. Addison-Wesley (Pearson Education), Upper Saddle River, NJ. 978-0768684964.

Liskov, B., Snyder, A., Atkinson, R., Schaffert, C. (1977). Abstraction Mechanisms in CLU. *Commun. ACM*, 20(8), 564–576. https://doi.org/10.1145/359763.359789

Lister, R., Simon, B., Thompson, E., Whalley, J.L., Prasad, C. (2006). Not Seeing the Forest for the Trees: Novice Programmers and the SOLO Taxonomy. In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITICSE '06. ACM, New York, NY, USA, pp. 118–122. 1-59593-055-8. https://doi.org/10.1145/1140124.1140157

Machanick, P. (1998). The Abstraction-First Approach to Data Abstraction and Algorithms. *Computers & Education*, 31(2), 135–150. https://doi.org/10.1016/S0360-1315(97)00064-X

Martin, R.C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, USA. 978-0-13-597444-5.

Mason, J. (1989). Mathematical abstraction as the result of a delicate shift of attention. *Learn. Math.*, 9(2), 2–8.

Mirolo, C., Di Vano, D. (2013). "Welcome to Nimrod" to Learn CS Ideas in the Middle School. In: *Proceedings of the 8th Workshop in Primary and Secondary Computing Education*. WiPSCE '13. ACM, New York, NY, USA, pp. 61–70. 978-1-4503-2455-7. https://doi.org/10.1145/2532748.2532756

Muller, O., Haberman, B. (2008). Supporting abstraction processes in problem solving through pattern-oriented instruction. *Computer Science Education*, 18(3), 187–212. https://doi.org/10.1080/08993400802332548

Nicholson, K., Good, J., Howland, K. (2009). Concrete Thoughts on Abstraction. In: *Proc. of the 21th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, pp. 1–10. http://www.ppig.org

Ohlsson, S., Lehtinen, E. (1997). Abstraction and the acquisition of complex ideas. *International Journal of Educational Research*, 27(1), 37–48. https://doi.org/10.1016/S0883-0355(97)88442-X

Pea, R.D., Kurland, D.M. (1984). On the cognitive effects of learning computer programming. *New Ideas in Psychology*, 2(2), 137–168. https://doi.org/10.1016/0732-118X(84)90018-7

Peel, A., Sadler, T.D., Friedrichsen, P. (2019). Learning natural selection through computational thinking: Unplugged design of algorithmic explanations. *Journal of Research in Science Teaching*, 56(7), 983–1007. https://doi.org/10.1002/tea.21545

Perrenet, J., Groote, J., Kaasenbrood, E. (2005). Exploring students' understanding of the concept of algorithm: levels of abstraction. In: *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*. ITiCSE '05*: Vol.* 37. Association for Computing Machinery, New York, NY, USA, pp. 64–68. 1595930248. https://doi.org/10.1145/1151954.1067467

Philpott, A., Clear, T., Whalley, J. (2009). Understanding student performance on an algorithm simulation task: implications for guided learning. In: *SIGCSE '09: Proceedings of the 40th ACM Technical Symposium on Computer Science Education*. ACM, New York, NY, USA, pp. 408–412. 978-1-60558-183-5. https://doi.org/10.1145/1508865.1509012

Piaget, J. (1969). *Psychologie et Pédagogie*. Bibliothèque Médiations*: Vol.* 59. Gonthiers & Denoël, Paris, France. 2282300599.

Piaget, J., Inhelder, B., Inhelder, B., Kagan, J., Weaver, H. (1969). *Psychology Of The Child*. The Psychology of the Child. Basic Books, New York, USA. 978-0465095001.

Podolefsky, N.S., Finkelstein, N.D. (2007). Analogical scaffolding and the learning of abstract ideas in physics: An example from electromagnetic waves. *Phys. Rev. ST Phys. Educ. Res.*, 3, 010109. https://doi.org/10.1103/PhysRevSTPER.3.010109

Rijke, W.J., Bollen, L., Eysink, T.H., Tolboom, J.L. (2018). Computational thinking in primary school: An examination of abstraction and decomposition in different age groups. *Informatics in Education*, 17(1), 77–92.

Sakhnini, V., Hazzan, O. (2008). Reducing Abstraction in High School Computer Science Education: The Case of Definition, Implementation, and Use of Abstract Data Types. *J. Educ. Resour. Comput.*, 8(2), 5–1513. https://doi.org/10.1145/1362787.1362789

Salomon, G., Perkins, D.N. (1989). Rocky Roads to Transfer: Rethinking Mechanism of a Neglected Phenomenon. *Educational Psychologist*, 24(2), 113–142. https://doi.org/10.1207/s15326985ep2402_1

Scheiner, T., Pinto, M.M.F. (2016). Images of abstraction in mathematics education: contradictions, controversies, and convergences. In: Csíkos, C., Rausch, A., Szitányi, J. (Eds.), *Proceedings of the 40th Conference of the International Group for the Psychology of Mathematics Education -PME* (Vol. 4), pp. 155–162.

Sfard, A. (1991). On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*, 22(1), 1–36. https://doi.org/10.1007/BF00302715

Shute, V.J., Sun, C., Asbell-Clarke, J. (2017). Demystifying computational thinking. *Educational Research Review*, 22, 142–158. https://doi.org//10.1016/j.edurev.2017.09.003

Soloway, E. (1986). Learning to Program = Learning to Construct Mechanisms and Explanations. *Commun. ACM*, 29(9), 850–858. https://doi.org/10.1145/6592.6594

Spiro, R.J., Feltovich, P.J., Coulson, R.L., Anderson, D.K. (1989). Multiple analogies for complex concepts: antidotes for analogy-induced misconception in advanced knowledge acquisition. In: Vosniadou, S., Ortony, A. (Eds.), *Similarity and Analogical Reasoning*. Cambridge University Press, Cambridge, UK, pp. 498–531. https://doi.org/10.1017/CBO9780511529863.023

Statter, D., Armoni, M. (2016). Teaching Abstract Thinking in Introduction to Computer Science for 7th Graders. In: *Proceedings of the 11th Workshop in Primary and Secondary Computing Education*. WiPSCE'16. ACM, New York, NY, USA, pp. 80–83. https://doi.org/10.1145/2978249.2978261

Statter, D., Armoni, M. (2020). Teaching Abstraction in Computer Science to 7th Grade Students. *ACM Trans. Comput. Educ.*, 20(1). https://doi.org/10.1145/3372143

Verhoeff, T. (2011). On Abstraction and Informatics. In: Kalaš, I., Mittermeir, R.T. (Eds.), *Informaticsin-Schools. Contributing to 21st Century Education: 5th International Conference on Informatics in Schools: Situation, Evolution and Perspectives, ISSEP 2011, Bratislava, Slovakia, October 26-29, 2011. Proceedings*, pp. 1–12. 978-80-89186-90-7. https://doi.org/www.issep2011.org

Waite, J.L., Curzon, P., Marsh, W., Sentance, S., Hadwen-Bennett, A. (2018). Abstraction in action: K-5 teachers' uses of levels of abstraction, particularly the design level, in teaching programming. *International Journal of Computer Science Education in Schools*, 2(1), 14–40. https://doi.org/10.21585/ijcses.v2i1.23

Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., Wilensky, U. (2016). Defining Computational Thinking for Mathematics and Science Classrooms. *Journal of Science Education and Technology*, 25(1), 127–147. 1573-1839. https://doi.org/10.1007/s10956-015-9581-5

White, P., Mitchelmore, M. (1999). Learning mathematics: A New Look at Generalisation and Abstraction. In: *AARE Annual Conference*. AARE '99. Australian Association for Research in Education, deakin, ACT, Australia, pp. 1–12.

Wilensky, U. (1991). Abstract meditations on the concrete and concrete implications for mathematical education. In: Harel, I., Papert, S. (Eds.), *Constructionism*. Ablex Publishing Corporation, Norwood, NJ, pp. 193–203.

Wing, J.M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–35. https://doi.org/10.1145/1118178.1118215

Wing, J.M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society*, 366, 3717–3725. A / Math Phys Eng Sci. https://doi.org/10.1098/rsta.2008.0118

Wing, J.M. (2011). Computational Thinking: What and Why? *The Link Magazine*.

C. Mirolo is assistant professor in Computer Science at the University of Udine, where he currently teaches introductory programming, computational geometry and computer science education. He has also been responsible for the professional development programmes for high-school computer science teachers. His research interests include the learning of programming and the role of computational thinking in general primary and secondary education.

C. Izu is lecturer in Computer Science at the University of Adelaide. She has a degree in Computer Science and PhD in Computer Architecture. Cruz has been active in the area of Computer Science Education in the last 6 years, exploring computational thinking, and how to teach programming skills, problem solving and code comprehension to undergraduate students.

V. Lonati is assistant professor in Computer Science at the University of Milan. Degree in mathematics and PhD in computer science, her research interests include introductory programming learning, computing education at K-12 level, constructivist strategies in computing education, professional development for teachers. She is member of the Scientific Committee of the Italian Bebras Challenge and former member of the International Bebras Board.

E. Scapin is an experienced Computer Science teacher at the I.T.T. "G. Chilesotti" in Thiene (VI). Currently, he is pursuing a PhD programme at the University of Udine, with a research project in Computer Science Education. His main topic of interest concerns task-related models to improve the learning of programming, and more specifically iteration, at the upper secondary instruction level.