What We Talk About When We Talk About Programs

Violetta Lonati*
Università degli Studi di Milano
Dept. of Computer Science, Lab. CINI "Informatica e Scuola"
Milan, Italy
violetta.lonati@unimi.it

Andrej Brodnik*
University of Primorska
Koper, Slovenia
University of Ljubljana
Ljubljana, Slovenia
andrej.brodnik@upr.si

Tim Bell
University of Canterbury
Christchurch, New Zealand
tim.bell@canterbury.ac.nz

Andrew Paul Csizmadia
Newman University
Education
Birmingham, United Kingdom
a.p.csizmadia@newman.ac.uk

Liesbeth De Mol
CNRS, Université de Lille
UMR 8163 Savoirs, Textes, Langage
Lille, France
liesbeth.de-mol@univ-lille.fr

Henry Hickman
University of Canterbury
Christchurch, New Zealand
henry.hickman@pg.canterbury.ac.nz

Therese Keane
La Trobe University
Melbourne, Australia
t.keane@latrobe.edu.au

Claudio Mirolo
University of Udine
Dept. of Mathematics, Computer Science and Physics, Lab. CINI "Informatica e Scuola"
Udine, Italy
claudio.mirolo@uniud.it

Mattia Monga
Università degli Studi di Milano
Dept. of Computer Science, Lab. CINI "Informatica e Scuola"
Milan, Italy
mattia.monga@unimi.it

ABSTRACT

Programming plays a paramount role in many educational policies and initiatives. However, the current focus on coding skills poses the risk of giving students an overly simplistic and impoverished idea of what programming means and involves. Their experiences would be much more significant if learning encompassed understanding the richness of the nature of programs.

Programs permeate our lives more inextricably than might often be recognised, and all citizens of the digital era could benefit from understanding their multifaceted nature. A fundamental component of such an understanding is getting a sense of how programs are created and work (i.e., the programming process).

But programs are strange creatures that escape simple definitions. To the best of our knowledge, there is no Nature of Programs framework that teachers and policy makers can use to shape their practice and targets. In this working group we developed such a framework, by collecting and organising contributions from the literature and the computer science education community. The paper presents the framework and the rationale behind its development. It is anticipated that the framework can be used to inform the design of sound curricula, and support teachers and learners to understand the bigger picture around programming.

CCS CONCEPTS

• **Theory of computation** → **Design and analysis of algorithms**; • **Social and professional topics** → *Computing education.*

KEYWORDS

programming, nature of programs framework, computer science education

ACM Reference Format:
Violetta Lonati, Andrej Brodnik, Tim Bell, Andrew Paul Csizmadia, Liesbeth De Mol, Henry Hickman, Therese Keane, Claudio Mirolo, and Mattia Monga. 2022. What We Talk About When We Talk About Programs. In *2022 ITiCSE Working Group Reports (ITiCSE-WGR '22), July 8–13, 2022, Dublin, Ireland.* ACM, New York, NY, USA, 48 pages. https://doi.org/10.1145/3571785.3574125

1 INTRODUCTION

> *We still think too readily of programs as just being for compilation. We should think of them also as being for communication from ourselves to others, and as vehicles for expressing our own thoughts to ourselves.*
>
> Thomas R.G. Green (1990) [61]

> *Because software seems to be an intangible intellectual product we can colour it to suit our interests and prejudices. For some people the central product of software development is the computation evoked. For some it is the social consensus achieved in negotiating the specifications. For some it is a mathematical edifice of axioms and theorems. Some people have been pleased to have their programs described as logical poems. Some have advocated literate programming. Some see software as an expression of business policy.*
>
> Michael Jackson (1985) [78]

*co-leader

ITiCSE-WGR '22, July 8–13, 2022, Dublin, Ireland

ACM ISBN 979-8-4007-0010-1/22/07…$15.00
https://doi.org/10.1145/3571785.3574125

Programs are all around us, and they are becoming a significant part of daily life; they are relied on to wake us up, check the weather, communicate with friends and colleagues, plan our transport, order food, and are essential tools in many forms of employment. The significance of computer programs in daily life is reflected, for example, by the inclusion of programming as a key digital competence in the DigComp Framework [31], and teaching programming in K-12 education has been widely promoted [51].

Programming has been taught in tertiary institutions for decades, and is part of a well-established curriculum with clear vocational pathways. However, this is often not the case in K-12 education. There is a growing mandate in some countries for programming to be taught at grade-school level, but in other countries the teaching of programming is ad-hoc and relies on the motivation and expertise of teachers to deliver it — see, e.g., [51, 73]; in particular, based on an in-depth analysis of K-12 computer science education in a number of case studies from heterogeneous countries, Hübwieser and colleagues conclude that "programming in one form or another, seems to be absolutely necessary for a future oriented" computer science education [73].

Where programming is mandated in the curriculum, it is often delivered by teachers who don't have such intrinsic motivation to deliver it, and so the importance of understanding what it is and why it is significant is essential. Learning to program and participating in the digital economy opens many career opportunities, and assists students to understand and participate in our digital world. Learning to program can be seen as a practical approach to understanding what a program is, but if the focus is *only* on being able to program, students can miss the forest for the trees — they can fail to see the connection between their classroom exercises and the software that permeates their lives.

Thus, although programming has become a significant topic in education, many teachers may not feel sufficiently familiar with it [50]. The current generation of teachers is indeed unlikely to have experienced programming when they were at school. Moreover, there is terminology such as "programming", "coding" and "computational thinking" that can be confused with each other (see e.g., [33]), and their meanings have evolved as they are used in an educational context. The terms computer programming, coding and computational thinking are all situated in computer science and are characterized by engagement in an analytical process of problem-solving. Traditionally "coding" has referred to the narrow part of programming that involves converting the idea for a program into the syntax of a programming language (the "source code"), whereas it is now often used in schools as being equivalent to the whole process of developing a program[1]. For many school-based educators, there is no difference between the terms "programming" and "coding" even though technically there is considerable gap between the two. Likewise, computational thinking has been described from many points of view [35], and often is positioned as having programming as a core activity. Even taking Wing's fairly broad definition of computational thinking as "the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent" [162], we can see that programming certainly exercises many of the key ideas of computational thinking because it works with an information-processing agent (a computer receiving instructions written in a programming language) that behaves consistently and can be invoked over and over to test if the student's thinking (expressed in the program) gets the desired result.

One extreme view that has been expressed is that terminology such as "programming" and "coding" are sufficiently confusing and that we should really refer to these disciplines using a term such as "problem solving" [157]. However, changing the programming/coding terminology would be almost impossible now because it is now so ingrained in books, curricula, and even the names of organisations such as Code Club and Code.org. Adding to the confusion is that students may not recognise that programs they are learning to write have the same *nature* as the "apps" on their phone, the software that runs a "smart TV", a download that they install on a smartwatch, an "update" to firmware, or the software that controls the fuel injection in turbo charged cars. Learners will often start in simpler environments (such as block-based languages controlling animations or simple robots), or work with solving small problems that can be programmed within the space of lesson (such as adding up a list of prices). Often the software that is used by consumers on a daily basis is created by large well-resourced teams of developers, although despite this happening in a different environment to the classroom, both the learner and professional contexts involve exercising the same skills, and the fundamental nature of what they are doing is the same. If a student were to look for a definition of programming, they can be confronted with many different interpretations of what a program is, and the reasons for writing them. Programs have been variously described as a tool for problem solving [157], an abstract symbol manipulator [41], the result of a collaborative effort [166], and can even be viewed as a work of art in their own right [164]. To overcome focusing on one limited perspective, teachers need to be aware of what the nature of programs (NoP) is, and use this knowledge to inform their teaching practice.

In this working group we explored this multifaceted nature of programs, and developed the document that appears in Appendix A, from now on referred to as "the NoP framework". We aimed for a scientifically sound description of what a program is, having in mind that this can help learners not as much to be trained to write programs, but rather to build the big picture about programs. The contribution of the NoP framework is to provide an integrated holistic presentation of the basic elements of the nature of programs, synthesised from the many points of view that have been expressed in the literature about the nature of programs.

Given the purposes of the current work, we restrict ourselves to programs that one could call "traditional". That is, our focus is on programs which are implementation of human designed algorithms, which consequently allows us to explain how they achieve their intended goal. Such a decision leaves out of our scope programs whose implemented logic is not human made but machine produced (e.g. by evolutionary programming or machine learning techniques). For a more detailed description of this distinction, see Section 3.2.4.

In order to develop the NoP framework, the research team used an iterative process where the team pooled ideas together through

[1]For example, see the introduction to [84].

regular collaborative virtual weekly meetings. This involved looking at a variety of models that have been proposed for describing programs and programming, and developing a way to unify them. A literature review was performed by searching for relevant terms in the computing literature, but also by conducting an exhaustive manual search of major computer science education publication avenues from the last few years. This informed the development of the framework, which resulted in a draft version that was shared on relevant mailing lists, in order to seek feedback from experts and practitioners in computer science education. This feedback was analysed and aggregated and was used to improve the document. Appendix A contains the NoP framework after this revision.

It is anticipated that the NoP framework can be used to inform the design of sound curricula and support teachers and learners to understand the bigger picture around programming. In the remainder of this paper we describe the background and reasoning that lead to the framework, including combining literature reviews and input from experts. It is written for school teachers and policy makers to assist them to see where programming belongs and the rationale of why and how it should be included in the curriculum. Because of the widespread use in education of the term "coding" to mean "programming", we allude to both of these terms in the NoP framework, so that it can be recognisable to educators. The nature of programs will be examined from six facets (points of view), focusing on the properties that most programs have, although there will always be cases on the boundary line of what might be regarded as a program, and how it could be perceived. For each of the facets, examples and impacts are provided to illustrate how they might be encountered in practice.

It is worth noting that, although the focus of this work is on programs, this is not to downplay the many other aspects of computing that are important for students to engage with. Within computer science, programming is just one of many concepts that are covered to prepare students to work in the field, and in broader curricula (particularly in K-12 schools), understanding programming is just one aspect of the digital competencies that students need as digital citizens. In computer science, programming is an underpinning skill that is applied to many aspects of the discipline. In a broader context, such as K-12 education, it has a qualitative difference from other computing topics because it is about *creating* rather than *using* digital technologies [99].

The paper is organized as follows. We first review the literature in Section 2. In Section 3 we illustrate the framework we developed and the rationale behind it. Section 4 presents the feedback we have collected from educators and computer science experts, and reports on how the framework was improved accordingly. Section 7 provides a conclusion.

2 BACKGROUND

In this section, after summarising the methodology underlying our exploration of the literature, we review perspectives that have been expressed in the literature about what programs are, including definitions that have been proposed (Section 2.1), different ways programs can be understood (Section 2.2), non-technical angles relating to programs (Section 2.3), roles of programs in computer science education (Section 2.4), as well as some more critical attitudes towards (mandatory) computing and coding in K–12 education (Section 2.5). As this section spans a broad range of topics that can be related in some way to the nature of programs, after reading Section 2.1, which is the most relevant for the scope of the Framework, the reader interested only in the Framework per se may want to skip directly to Section 3.

Methodology. To begin with, two main difficulties are encountered in order to identify relevant contributions that address the nature of programs. First, the search could not be simply based on a short list of specific keywords, such as "nature" or "characterisation", in connection with "program". In fact, the nature of programs can be described using several different terms and, as a consequence, we have been unable to rely largely on automatic means for filtering a significant bulk of bibliographic items. Second, the literature is much more concerned with "programming" — especially technical aspects of programming — than with "programs"; when dealing with programming, the meaning of program is often taken for granted, maybe implied implicitly, but usually just left indeterminate. Thus, most of the "filtering" work has required direct inspection (by the authors) of large bibliographic datasets.

For these reasons, in particular the second point, our literature review is unlikely to be really complete. Nevertheless, having drawn from the diversity of the group to bring together many different sources, we think that the outcome is representative of the diversity of existing characterisations of the nature of programs and programming. To be more specific, in light of the above remarks, the review process included the following stages:

- Systematic search relative to major computer science education (and somehow related to the topic) conferences and journals, looking first at the title (for pertinence), then at the abstract and keywords (mentions of program/programming), and finally at the content of promising items.
- Forwards and backwards snowballing, starting from the references found in the previous stage as well as from more foundational sources known by the authors, to identify other relevant contributions.

For the first stage, we explored the following sources. Journals: ACM TOCE (all: 2001–2022), ACM Inroads (2018–2022), Informatics in Education (all: 2002–2022), Computer Science Education (2018–2022), CACM (2018–2022), Computers in Education (2018–2022), Computers and Education (2018–2022), Education and Information Technologies (2018–2022). Conference proceedings: ICER (2018–2022), ACE (2018–2022), Koli calling (2018–2022), WIPSCE (2018–2022), ISSEP (all: 2005–2021), FIE – Frontiers in Education (2018–2022), VL-HCC (all), WCCE (2018–2022).

2.1 Definitions of programs through the history and philosophy of computer science

> *The history of software is the history of how various communities of practitioners have put their portion of the world into the computer. That has meant translating their experience and understanding of the world into computational models, which in turn has meant creating new ways of thinking about the world computationally and devising new tools for expressing that thinking in the form of working programs.*

Michael S. Mahoney (2008) [105]

Since programs are one of the basic objects of study in computer science, and programming is a core activity, one might expect that within computer science itself there would be a clear and accepted definition already of what a program is. Indeed, from the late 1940s to the present day, one can find numerous definitions, either explicitly in glossaries, or implicitly in research papers, viewpoints, reports and programming manuals. For instance, one influential definition was provided by Grace Hopper in the 1954 *First glossary of programming terminology of the ACM* [70], which was then picked up in several other glossaries:

> **Program** (noun) - a plan for the solution of a problem. A complete program includes plans for the transcription of data, coding for the computer and plans for the absorption of the results into the system. The list of coded instructions is called a **routine**.

Another common definition referred to programs as the scheduling and sequencing of coded instructions, and was embedded in a much older engineering tradition of so-called program devices. It is from there that the notion of program was introduced around ENIAC, one of the first digital computers, and the origin of our current notion of program [36]. That understanding, program as a sequence of written instructions, was picked up in the context of early high-level programming languages and in particular the practice around ALGOL:

> Sequences of statements and declarations, when appropriately combined, are called programs [122].

In the 1950s and early 1960s the range of applications for computer programming was still rather limited, focusing mostly on scientific and business computing. The need for commercial software was only beginning, and programs were far from being the ubiquitous objects they are today because computer hardware was only available to large organisations. The history of programs, software, programming and programming languages has been studied to some extent already in the historical literature and we mention a few sources here. In [124], a long-term history of programming notations and languages is offered from the perspective of formal notations. Ensmenger [48] offers a study of the programming field from the viewpoint of the changing views and problems related to the profession of programming. The in-depth study [66] of programming practices around ENIAC introduces a notion of the so-called modern code paradigm to replace the historically problematic notion of stored program. Finally, the proceedings of the *History of programming languages* remains an important source to understand the various concepts and views on programming frameworks. Other views of programming from a historical perspective are advanced in [48, 65, 125].

The way we make and use programs today has evolved from the way computer programs were made and used in the early days of computing, and so older definitions such as Hopper's don't relate to the way students would see programs currently. Still, it is clear that already from the 1950s onward there were certain conceptions of computer programs that are still highly popular today amongst computer scientists.

The nature of programs also plays a key role in the broader discussion about the nature of Computer Science, starting from some earlier significant contributions that appeared in the '80s and '90s [38, 88, 137, 154, 159], one of the triggering question being *"is computer science a science?"* More recent contributions in this respect can then be found in [12, 29, 39, 71, 79, 127, 146, 147]. From these debates it is clear that the field of computer science itself does not have a clear identity and can at best be understood as a multidisciplinary domain. This is reflected in the diversity of conceptions of computer programs and programming that one finds in the literature, where often only one aspect of programs is emphasized.[2]

For instance, a programming language designer will probably have a definition that looks at programs primarily from a notational perspective, considering for instance the problem of formal semantics; a theoretical computer scientist might view programs as abstract procedures and declarations; while a software engineer might focus more on the complexity of computer programs and their relationship to the real world. This situation has resulted in a number of different views on what computer programs are. For instance, Donald Knuth has provided definitions focusing on programs as implementations of algorithms, and on the notational aspect of programs:

> An expression of a computational method in a computer language is called a program [87, p. 5].

Interestingly, later Knuth also provided quite a different definition of programs, anchored in the idea that "*[i]nstead of imagining that our main task is to instruct a computer what to do, [we should] concentrate rather on explaining to human beings what we want a computer to do* [89]." His proposal from that perspective was to view programs as "works of literature", hence the approach that came to be known as literate programming.

Scholars like Edsger Dijkstra emphasized the mathematical nature of computer programs even further by saying that a program is:

> an abstract symbol manipulator, which can be turned into a concrete one by supplying a computer to it [40].

James Fetzer, a philosopher and one of the main actors in the so-called formal verification debates of the 1980s [146] and who strongly opposed the mathematical viewpoint of Dijkstra, emphasized the distinction between programs as abstract mathematical objects and programs as causal models (due to their physicality). This led to the following definition [53]:

> From a methodological point of view, it might be said that programs are conjectures, while executions are attempted — and all too frequently successful — refutations.

That is, a written program first needs to be executed before one can be sure that it will behave correctly. From an engineering perspective then, programs should not be reduced to their notations nor to some kind of mathematical structure. Instead, what this view emphasises is the idea of programs being part of a larger system. As Denning wrote in 2004, programs are:

> components of complex systems that must be designed under severe constraints [37].

[2]The fact that computer science itself has become ubiquitous in a number of other scientific fields has probably not helped in these debates.

As this short sample of program definitions shows, one's understanding and conception of programs is usually a confirmation of one particular view on computer science. Consequently, just like for the computer science discipline itself, what we have is a set of very diverse definitions of programs each highlighting different program facets. This fact was very much acknowledged by Michael Jackson when he wrote about software [78]:

> Because software seems to be an intangible intellectual product we can colour it to suit our interests and prejudices. For some people the central product of software development is the computation evoked. For some it is the social consensus achieved in negotiating the specifications. For some it is a mathematical edifice of axioms and theorems. Some people have been pleased to have their programs described as logical poems. Some have advocated literate programming. Some see software as an expression of business policy.

But then added his own position:

> But many people here will surely want to think of software development as a kind of engineering.

Essential in Jackson's view is that software engineering is concerned with the relation between the world and the machine identifying different facets of that relation: modeling, interface, engineering and problems. The engineering facet is captured by the trinity of requirements, program and specification which was a core component of the model proposed later in [62]. It is then thus not surprising that in computer science education we have different views about how programming should be taught, and what the goals are. Indeed, perhaps this is one part of the explanation of the growing conflation of terms like coding, computational thinking and programming that we discussed in the Introduction.

Other authors have proposed program classifications that result in a more diversified understanding of computer programs. A well-known example from software engineering is Lehman's work on the basic laws of program evolution, which proposes a classification of programs in terms of their complexity and entanglement with the world [100]. The question of definition and classification is a traditional one for analytical philosophers too. Just to mention some, a central idea of Colburn's essay [27] is the peculiar role of abstraction in computer science and programming. Eden [46], on the other hand, discusses the contrasting ontological, methodological and epistemological characterisations of programs according to the perspectives of mathematics, engineering and science. In a similar way, Irmak [76] compares programs to music, emphasizing that just like music, programs too have a dual nature, and concluding that programs are abstract artefacts. The idea of programs having a dual nature is recurring not only in the philosophical literature but can also be found in programming, computer science, and computer science education. Alan Kay for instance wrote:

> Intangible message embedded in a material medium is the essence of computer software [...]. Is the computer a car to be driven or an essay to be written? Much of the confusion comes from trying to resolve the question at this level. The protean nature of the computer is such that it can act as a machine or like a language to be shaped and exploited [83].

In Turner's book [155] programs are understood through the lens of the philosophy of technological artefacts, based on the structure/function dichotomy, and of the design process; more specifically, in his view programs are defined by the "trinity" of specification, symbolic program, and physical process.[3] Finally, we mention here the work on software ontology developed in the setting of semantic web technology [96] presenting a so-called *Core Ontology of Programs and Software*. That work is anchored in the analysis by Eden and Turner mentioned above.

Recently there have been a number of collaborations that acknowledge the need for a multifaceted approach. The work of [19] starts out from the observation that "programs" are complex objects with different facets.[4] They are real — they affect our real lives; they are abstract — they process abstract entities; and they are concrete — they take up space in digital devices' memory, and can be copied, transferred, and corrupted. The recent PROGRAMme project[5] also starts out from the question "What is a computer program?", and proposes an analysis of programs in terms of their different modalities: physical, socio-technical and notational. However, that project is mostly aimed at historians, philosophers and computer scientists. While issues of literacy and the growing lack of understanding of programs by the layperson is surely a driving motivation in that project, it does not propose a framework that is specifically focused on computer science education.

From these many points of view, we can conclude that while it is fair to say that there has been some acknowledgement in the literature of the multifaceted nature of computer programs, a more holistic approach is very much needed if the aim is to provide a *general* literacy, or as Alan Kay put it:

> To get the medium's magic to work for one's aims rather than against them is to attain literacy [83].

Such general literacy can only be achieved if we are prepared to embrace the full reach of computer programs.

2.2 Programs and programming

> *[J]ust a few building blocks suffice to enable us to write programs that can help solve all sorts of fascinating, but otherwise unapproachable, problems. [...]*
>
> Robert Sedgewick & Kevin Wayne (2008) [135]

According to Pennington & Grabowski [121], "Programming problems are unique in that they usually involve solving a problem in another (application) problem domain, such as mathematics, accounting, electronics or physics, in addition to solving the program design problem." In addition, in the opinion of some educators, e.g. [17, 60], the nature and status of programming are difficult to characterise. In an interesting section titled "What is a program?", Pair [116] introduces a few possible conceptions of *program*, based on syntactical vs. semantic views, as well as on the implications of different programming paradigms.

The structure of programs can also be understood in terms of more abstract patterns than those directly offered by the programming language constructs. Despite the infinite variations in how

[3] For a review on the existing philosophical literature on the ontology of computer programs, see [126].
[4] In fact, this paper was one of the driving forces behind the current paper.
[5] See: www.programme.hypotheses.org

programs can be written to achieve the same thing, we can indeed recognise some recurrent patterns (and anti-patterns) that have been identified since they can provide a strong basis for the organisation of programs. They have also been recommended for teaching to beginner programmers so that they are aware of common techniques. The idea of *design patterns* is based on the work of Alexander [3], and the most well known are from the "gang of four" [58], intended for OO programming.

Cunningham [34] highlights the benefits of seeing programming as making use of patterns (schemas) rather than a combination of syntactic elements: "There are endless ways to combine syntax elements, and a correspondingly large mental search space for the novice programmer to think through when writing code or understanding someone else's code." In addition, she remarks that "[w]hile rare in classrooms, this approach potentially aligns with a basic psychological fact: humans often use schemas (mental patterns or frames) to organize their knowledge." Elementary patterns, conceived for the benefit of beginner programmers and applying in non-object-oriented environments, have been proposed since the late 90s by a number of educators, in particular [7, 14, 26, 158], and more recently Amanullah [5] has also considered the use of elementary patterns to analyse Scratch programs.

Among the several features that can be associated to programs, Guzdial and Ericson [64] emphasise the role of *names*: "Much of programming is about naming: A computer can associate names, or symbols, with just about anything [...]. A computer scientist sees a choice of names as being high quality in the same way that a philosopher or mathematician might: If the names are elegant, parsimonious, and usable."

Some authors, such as Park & Kim [120] and Utting et al. [156], view programming as a new form of basic literacy that should be part of a compulsory curriculum offering. Supporting this argument, Burke [21] aligns programming as both writing and storytelling, which has created "a new and unique form of digital composition emerging over this century in which words, images and sounds are not only arranged as text but coded sequentially as a unified narrative." Whilst Burke, O'Byrne & Kafai [22] and Kafai [82] regard teaching programming as a literacy, especially at a young age, they also discuss a flow-on effect, allowing children to better articulate thoughts and connect ideas together: "Programs are personal objects that can be shared publicly". These authors envisage "a shift from computational thinking to computational participation across four dimensions: a shift from code to actual applications [...]; a shift from tools to communities [...]; a shift from starting from scratch to remixing [...]; a shift from screens to tangibles [...]."

A human-computer interaction paradigm which is recently receiving increasing attention is *literate computing* and views programs — including professional programs — as part of a more comprehensive "narration", see e.g. Fog and Klokmose's review [55]. In these environments, *literate programs* combine executable code together with a mix of texts and other media that are dynamically and interactively manipulated by the code. Thus, code, the processed objects and the produced output, all reside in the same space, so dissolving "the traditional distinction between programming and using computers, but also between using and developing software tools" [55]. Literate programming is also considered for its educational potential. Roy, for instance, proposes to extend the scope of programming languages addressed to novices in order to improve the expressive power and make this enriched documentation "an integral part of the programming process" [129].

Finally, in a couple of witty pieces [114, 115], Nobel and Biddle see a *postmodern* trend in the programming practice and raise the question of what programming should be about from that perspective. In fact, the new forms programming is taking may also blur the notion of program.

The varied perspectives mentioned above highlight structural, expressive, literacy-related aspects of programs and programming. Although the summary is not exhaustive, it testifies to the diversity of ways in which programs can be conceptualised and suggests that any possible characterisation of their nature, including of course the framework we propose in Appendix A, will inevitably emphasise some aspects while understating others.

2.3 Psychological, social and ethical concerns related to developing and using programs

> *[C]omputing is a technology [...]. The question is not whether new technology involves social change, but how it does. In particular, it is a question of agency. As a form of technological determinism, the impact theory leaves people reacting to technology, rather than actively shaping it.*
>
> Michael S. Mahoney (2005) [104]

Programming, or even just using programs, have cognitive and psychological implications for the individual engaged in a task, as well as societal implications for the involved community, and these affect the way the nature of programs is perceived. As a consequence, the program as a technical artefact, seen in isolation, does not fully determine its meaning, but we need to consider also the context, including people's related endeavours and expectations. In the following, we summarise a few reflections in this respect, which are especially relevant for educational purposes.

As part of a monograph that collects early studies taking a psychological perspective, Green [61] discussed a range of cognitive-psychological issues in connection with the nature of programming. Then, in this vein, Blackwell [15] asks "what is distinctive about the cognitive tasks involved in programming, and in particular which distinctive cognitive tasks are shared between all programmers, whether professional or end-users." And he posits that this question should be addressed without questioning if, in the end-user case, the "activity is genuine programming", but simply by analysing "their experience in order to understand the general nature of programming activity."

More specifically, Blackwell identifies three cognitive features of programming (with clear implications as to the nature of programs):

1. *Loss of direct manipulation* – "The situation in which the program is to be executed may not be available for inspection, because it may be in the future, or because the program may be applied to a greater range of data situations than are currently visible to the programmer."
2. *Use of notation* – "This is also a universal characteristic of abstract thought. [...] Abstraction results from forming some representation of the state of the world" and the "correspondence between a representation and the state of the world is one of convention, not of causality."

3. *Abstraction as a tool for complexity* – "Abstraction use in which the user conceptualizes common features of complex behaviour, then formulates notational abstractions in which to express them."

Moreover, in a related paper, Blackwell suggests "moving [...] away from conventional programming and software engineering to focus on the characteristic cognitive tasks of programming, whether or not they occur in a social context that would normally be called programming" [16]. And, incidentally, among the non-professional approaches to programming, Iacovides and Green [74] identify the "DIY-for-fun" approach, contingent and often enjoyed because of "the experience of challenge, breakdown, and breakthrough, just as in gaming." In similar scenarios, the development of programs is subject to less severe constraints and presents very different features when compared with software engineering processes.

Taking the program *reader* perspective, Tenenberg [148] focuses on the "economic choices" made "[i]n order to construct meaning from a program," which "are influenced not only by cognitive constraints but also by the organizational and social context in which the program-related activities occur." As a consequence, also the reader's beliefs about the knowledge (of the programming as well as the application domains) he/she shares with the program writer come into play.

While most instructional goals are focused on technical aspects of programming, "[t]he system perspective alone," as pointed out by Schulte [133], "fails in bringing together social and technical aspects" and is is more likely to reinforce, instead, their separation. So, according to Schulte, a *socio-technical* perspective elaborating on the duality structure/function has the potential to connect "*technology* (in its isolated, single-sided meaning) with individual and social experiences and practices." Other authors see programming as a socio-technical undertaking from a variety of persepctives, e.g. [16, 91, 166]. In particular, social skills are important in successful large-scale software development [166], and several "skills of a professional programmer are related to the social context rather than the technical one — writing and interpreting specification documents, participating in design meetings, estimating effort and so on" [16].

The programmer's experience also includes emotional and related dimensions, such as affects, attitudes, beliefs, or perception of self-efficacy, dimensions that can be explored through the lenses of the theories and models reviewed by Malmi and colleagues [106]. According to them, indeed, a "holistic understanding of learning programming is needed if we are to design and support the learning process in meaningful ways."

In education, stereotypes and personal understanding can affect a student's attitude to whether programming is something they could engage in; for example, the idea of a "geek gene" (that some people are inherently born as programmers) is a commonly held view despite being widely refuted [2, 49, 63, 101, 102] (although there can be significant differences between outcomes for students in programming courses, and in the time taken to write programs). Even the misconceptions that parents have about programming can have a flow on effect onto how well their children do in computer science [81].

Broadening the scope, programs, as Mahoney [104] remarked in his paragraphs following the above quote, in that technological artefacts "are not natural phenomena but the products of human design, that is, they are the result of matching available means to desired ends at acceptable cost." And although the "available means ultimately do rest on natural laws, which define the possibilities and limits of the technology, [...] desired ends and acceptable costs are matters of society." However, what is peculiar of computer programs is that "whereas other technologies may be said to have a nature of their own and thus to exercise some agency in their design, the computer has no such nature. Or, rather, its nature is protean; the computer is [...] what we make of it [...] through the tasks we set for it and the programs we write for it."

Labour shortages, and the drive for companies to become leaner and more efficient organisations, have resulted in many companies adopting automation technology, but with mixed outcomes [95, 120]. With computing technologies impacting every aspect of society, including commerce, entertainment, education, and health care, not only should we ethically consider the positive impacts, but also consider potential "disasters" and the negative and undesirable impact of these technologies [69, 97]. Yadav, Heath & Drew [163] argue that society, and in particular the computing education community, should not simply focus on the development of technical skills within both the existing and potential workforce, but on supporting communities, fostering criticality, and promoting ethical digital citizenship for all [95].

2.4 Role of the nature of programs in computer science education

> *The computer revolution is a revolution in the way we think and in the way we express what we think. The essence of this change is the emergence of what might best be called* procedural epistemology *— the study of the structure of knowledge from an imperative point of view, as opposed to the more declarative point of view taken by classical mathematical subjects.*
>
> H. Abelson, G.J. Sussman & J. Sussman (1984) [1]

Educators have written much about programming that is aimed at understanding the role of writing programs in an educational context. Usually, programming languages and programming environments for teaching how to program have their designs rooted in both Constructivism and Constructionism. Constructivism, a cognitive theory, was developed by Jean Piaget. Piaget's idea was indeed that knowledge is constructed by the learner and that learning is deriving meaning from the compilation of complex knowledge structures. Constructionism is an educational method, which is based on the constructivist learning theory, and was developed by Seymour Papert — a student of Piaget. Papert maintained that learning occurs "most felicitously" when constructing/creating a public artefact (digital or physical) [119], in our case a program.

When discussing technologies that support learning and education, Papert would often use the metaphor of "low floors and high ceilings." For a technology in general, and specifically for a programming language to be effective, it should provide novices with the opportunity to get quickly started (low floor) but also to have the opportunity to create increasingly sophisticated programs over time (high ceiling). Resnick and colleagues added another dimension to this analogy, that of "wide walls" [128]. This additional

dimension avoids to have a single path from low floor to high ceiling and provides multiple pathways that novice programmers can explore as they move from floor to ceiling.

While programming is widely considered a fundamental activity to understand computing (see e.g. Ben-Ari's "defense of programming" [13]), things can get even more involved, from an educational perspective, in that programs are often seen as an effective, concrete instrument to foster, on the one hand, the learning of more abstract computer science concepts — concepts introduced in accordance with the familiar motto "computer science is more than programming" [6] — and, on the other hand, for its transfer potential of problem-solving skills that can be spent in other fields.

On the other hand, many textbooks about programming give understandably simple definitions, or even leave the definition of the subject of the book to be inferred by the experience the reader has while they learn to create programs. Thus, a typical characterisation of programs is often limited to generic descriptions of few words, such as the following one from a textbook addressed to high school students [42] (translated from French):

> A program is a text that describes an algorithm that you want a machine to execute. This text is written in a particular language, called *programming language*.

Kelleher and Pausch [85] reviewed various programming environments, intended for instructional purposes, which convey different views of programs. In particular, to mention some, programs can be seen as:

- Combinations of iconic templates;
- Assemblies of *tangible*, physical programming bricks;
- Implicitly represented sequences of recorded, usually graphical, actions;
- Instances of *programming-by-demonstration*, where the algorithms are heuristically extrapolated from examples;
- Combinations of graphical rewrite rules, including conditions, actions, as well as *analogies* playing the role of more intuitive forms of code reuse;
- Finite-state machine diagrams, where each state can be edited;
- Data-driven flow diagrams;
- Hierarchical collections of procedures.

Some of the environments considered are conceived for *teamwork* and, from our perspective, raise the question about the nature of a program that can change dynamically through the interactions of team members.

According to Kelleher and Pausch's definition of programming in the educational context [85], a program is characterised as "a set of symbols representing computational actions," which "express [a user's] intentions to the computer" and allow "a user who understands the symbols [to] predict the behavior of the computer." The authors also remark that their definition does not cover *programming-by-demonstration* systems, where "the user cannot accurately predict what program will be produced," a problem shared by other kinds of "dark" programming techniques [80], a class of approaches that will be briefly discussed in section 3.2.4.

Having in mind childrens' cognitive abilities at early educational stages, Thompson and Tanimoto [151] establish relationships between six forms of programming activities — namely, requirements, specification, design, coding, debugging, reuse — and five levels of (cognitive) narrative development — labelling, listing, connecting, sequencing, narrating — that can be found, in particular, in storytelling. In their perspective, indeed, "[i]f children conceive of programs and stories in the same way or similar ways, then there may be value in teaching programming concepts in a way that mirrors the levels of narrative development."

One attempt to provide a multifaceted perspective on programs from an educational perspective is Schulte's [132] Block Model framework (Figure 1), which can be used to support program comprehension, e.g. [77]. In Figure 1 we can see a view ranging from notation-focussed elements (on the left) to the program as a tool for an end-user (on the right). The model also touches on the abstract nature of programs by highlighting the "algorithm underlying a program", at the same time acknowledging their executable nature (in the middle column). This latter relationship between abstract ideas and executability is described by Hromkovic, who explains programming as "translation of an algorithm into the computer language", but also contrasts it with the formal view that "this transformation can be viewed as a proof of the automatic executability of the algorithm described in a natural language" [72].

Du Boulay [43] identifies five "areas of difficulty" when learning to program that can be connected with elements of the framework we are going to introduce. Such areas are relative to: *orientation*, i.e. *what programs are about* in general; *notional machine*, the abstract model of the executing agent; *notation*; *structures*, which have to do with structure/function perspective on technologies; and *pragmatics*, which, in contrast, refers to programming activities. Du Boulay's elements highlight the importance of introducing students to different aspects of programming, and being prepared to address them specifically. The introduction of the concept of a *notional machines* highlights the executable nature of programs [44, 54, 139, 153].

Further obstacles to develop a clear understanding of programs are identified in their relationships with human thinking habits or, more in general, with the overall context in which they are meaningful. Weigend [160], for instance, focuses on the "barriers" that must be overcome when refining a first intuitive idea to come up with a really implementable solution. Tenenberg and Kolikant [149] suggest to view "computer programs as linguistic acts, as utterances that function with human communities." They then elaborate on this dialogical perspective, inspired by psycholinguistic studies of natural language, and propose to plan empirical research to investigate the extent to which novices may ascribe intentionality to the computer and are aware of the communicative function of their programs towards humans.

From a different angle, while analysing from an educational viewpoint the current role and shortcomings of the traditional notion of *programming paradigm*, Krishnamurthi and Fisler [93] contend that an alternative characterisation centered on behaviour — usually "captured in the concept of a *notional machine*" underlying the programming language — is more appropriate to make sense of programs. In support of their position, they argue that carefully chosen (to foster learning) notional machines are better suited than paradigm-related concepts to account for the program features in a range of programming models, such as *reactive*, *event-driven* or *embedded-systems* programming.

Looking at things from the students' point of view, Thuné & Eckerdal [152] have analysed novices' conceptions of programs,

(M) Macrostructure	Understanding the overall structure of the program text.	Understanding the *algorithm* underlying a program.	Understanding the goal/purpose of the program (in the context at hand).
(R) Relationships	Relations & references between blocks (e.g. method calls, object creation, data access...).	Sequence of method calls, *object sequence diagrams*.	Understanding how subgoals are related to goals, how function is achieved by subfunctions.
(B) Blocks (Chunks)	*Regions of Interest* (ROI) that syntactically or semantically build a unit.	Operations of a block, a method, or a ROI (chunk from a set of statements).	Understanding the function of a block, seen as a subgoal.
(A) Atoms	Language elements.	Operation of a statement.	Function of a statement: its purpose can only be understood in a context.
	(T) Text Surface	(P) Program Execution	(F) Function/Purpose
Duality	Architecture/Structure Dimensions		Relevance/Intention Dimension

Figure 1: Schulte's Block Model framework, from [77].

and found five qualitatively different dimensions that students need to discern when learning programming:

(1) the textual representation of a program
(2) the action of a program
(3) the application addressed by a program
(4) the problem to which the program is a solution
(5) the various contexts in which programming skills can be an empowering resource [152, Table 2].

The first dimension focuses on notation, the second on execution, and the remainder speak to a program as a human-made tool.

We also see facets of what programs are through the motivations for teaching programming. Learning to write computer programs is a fundamental part of most computing curricula, but there are a variety of reasons for introducing it (particularly at K-12 level), and a variety of ways that it is presented to students.

A commonly perceived reason for teaching the skill would be to give students vocational opportunities, and this becomes more accurate at higher levels of education. For example, Falkner points out that "There has been a push for the inclusion of computing curricula in schools to provide all children with an opportunity to develop fundamental skills in coding and computational thinking that can support future generations in the modern workplace" [52], and this is echoed by other authors e.g. [68, 113]. This goal shouldn't be underestimated, but the purpose of teaching programming at lower grade levels isn't so much as direct vocational preparation, but to inform students about what the nature of the whole discipline is so that they can make informed career choices:

> "Significant factors in the declining interest in computer science amongst school students in western countries include the widespread misunderstanding of the subject and the career, ... and the confusion of the discipline of computing with learning how to use the computer as a tool [11]."

Commentators are quick to point out that this isn't necessarily the main driver, particularly at lower grade levels, as there are other benefits to learning disciplines relating to programs and programming around developing their understanding and agency in the digital world [8] as well as increasing their general problem solving skills [131], and developing skills that are transferable across other disciplines [111, 130, 165].

2.5 Critical perspectives about computing and programming in K–12 general education

> *[S]ource code is a fetish [...]: we "primitive folk" worship source code as a magical entity — as a source of causality — when in truth the power lies elsewhere, most importantly in social and machinic relations. If code is performative, its effectiveness relies on human and machinic rituals.*
>
> Wendy Hui Kyong Chun (2008) [25]

Let us end this review with a summary of some more skeptical positions about computing in general education. The pressures and the "hurried" policies to (re)introduce computer programming in the K–12 curricula, starting in the late 2000s, have indeed also led to more critical reflections, in particular from the educational sciences' perspective.

30 years before, however, Papert had already recognised how "the complexity and subtlety of the computer presence" in schools made "it a challenging topic for critical analysis," in order to appreciate its implications on the human and social sides [118]. More specifically, Papert's major concern was about a "technocentric" view giving too much prominence to the technical instrument, e.g. LOGO, thought of as the source *producing* direct benefits on thinking and learning, while downplaying the role of what really matters in education, namely "people and cultures." According to Papert, the "advent of computers" required a re-examination of our long-established and often implicit assumptions about education. Learners, as well as teachers, should appropriate these emerging technologies as additional stuff "to do better whatever you are doing" — the focus is on what they are *doing*, not on the instrument itself.

More recently, one of the aspects thought to be subjected to critical reflection is the role played by economic/market factors in connection with the aspiration to empower learners as agents of change. For example, as Bresnihan et al. [18] crudely put it, from what they identify as a broadly Marxist stance, "the call for more computing in schools" might be simply ascribed to "capital seeking cheaper labour." In their analysis of the historical developments of computing in schools, Bresnihan and colleagues suggest that similar factors may also explain the decline of computing as programming and the transition to office applications in the 90s, under the pressures of "an industry hungry for trained office workers." Thus, while being inclined to support an educational agenda for computational thinking, they warn about the risk of "alienating another generation of young people" by paying little attention to the overall rationale for and pedagogy of computing education. It may

also be worth noting that the "economy-driven discursive framing of computing education" has been identified by Mertala [108] as a reason explaining teachers' reluctance towards computing, since it is perceived in contrast with their socialisation-oriented approach.

More generally, as pointed out by Dufva & Dufva [45], software technologies are not value-neutral, but incorporate "both conscious and subliminal values of the programmer, a software company or society's understanding of good code." Or, in Williamson's words [161], "Programming code captures ideas about how the world works and translates them into formalized models that can be computed through algorithmic procedures, which can then augment, mediate and regulate people's lives." Thus, "[h]ow we define [code] values and make sense of them is an essential part of coding education" [109]. Moreover, if "digital technologies and their uses in everyday life are not neutral, then," according to Burnett [23], "it follows that digital technology use in schools is not neutral either," but reflects beliefs about children's learning needs and hence "raises questions about the purpose and nature of schools themselves."

A more radical critique of how digital technologies and computational thinking are introduced in school, in particular at primary level, is given by Swertz,[6] e.g. [143]. He ascribes the current trends to *monistic*, "ideology"-driven perspectives about the future and what causes individuals to act, so envisaging the risk of transforming the medium from means to end of the instruction and of "getting people used to [...] serve computers." Following Humboldt, for Swertz the core of general education — better denoted by the German word *Bildung* — is the movement between non-overlapping languages, between the self and the world, which in his view can never be accomplished through digital media. In addition, by taking the pedagogical perspective of "future openness," the idea of Bildung means "above all to enable people to decide for themselves which area [e.g. of digital technologies] they want to develop and deepen in their lives and when."

The perspective about what *futures* are possible is also prominent in Mertala's analysis [108], who distinguishes between *functional* and *critical* paradigms in computing education. The former paradigm focuses on logical and algorithmic aspects of computing, including coding skills, where the main role of education is "to train students to be functional members of society *as it is*." The latter, on the other hand, "encourages students to criticize the prevalent societal structures and to act as agents of change," in the endeavour to shape the world "that *might be*." While technical and functional aspects seem to be dominant in computing education, according to Mertala the implied traditions should "be put under critical evaluation" by making the school practices "visible to the students" as well as "by switching between the often overlapping micro- [individual practices] and macro-level [societal implications] perspectives."

In the above respect, Pangrazio [117] points out the need to develop a critical disposition to understand "the role humans play in questioning, challenging and therefore shaping [the] technosocial system," in order to prevent "digital practices and tools from appearing as a series of *natural*, inevitable processes which become uncritically inscribed into daily life." In her view, individuals should be encouraged to move between technical and critical mindsets as part of their 'design' practices, including those implied in coding activities, by taking into account also how "discourse, ideology and power" are entwined in digital contexts. Likewise, Meyers [109] sees code as "a political technology" and adds that there is a need of "a strong ethical component to coding curricula that facilitates thinking through the consequences, intended or unintended," of programs from a socio-technical perspective. He then concludes that "perhaps the greatest lesson is [...] knowing when not to code."

Finally, Tamatea's analysis [144] focuses on the deepening engagement with digital "abstraction," meant in Baudrillard's terms [9], which is implied in coding tasks and can challenge "the tradition of liberal-humanism in education." For Baudrillard, indeed, the present-day overexposition to abstract ways to access the real carries the risk of deconstructing the notions of self, society and liberty. Tamatea identifies the predominant arguments deployed by the stakeholders supporting or opposing compulsory coding in schools. While most such arguments, from both sides, appear to be rooted in liberal-humanism, the supporters' "stronger commitment to children engaging in abstraction" may actually undermine liberal-humanist tenets, i.e. an educational perspective resonant with the idea of Bildung endorsed by Swertz. Computational participation, in particular, is regimented by some machinery, rather than arising in spontaneous forms which involve rich sensory interactions, and "through computational thinking, our categories of thought become those of the abstraction-based coding language we work with."

Some of the concerns raised by the scholars surveyed above are also considered in the NoP framework, while discussing technological artefacts as *tools* and *human-made* — see Appendix A.

3 THE FRAMEWORK

The main product of this working group is the NoP framework reported in Appendix A, which is a description of the Nature of Programs based on viewing the concept of a program through different facets, and on a concept map that describes the relationships between the elements that lead to a program being produced.[7] As explained in the Introduction, the discussion is restricted to traditional programs only.

Section 3.1 describes the process and the considerations that have contributed to define the contents of the framework. In Section 3.2 we outline the facets we use to characterise the nature of programs, while also justifying why we chose those particular facets to define the nature of programs. Then, in Section 3.3 we discuss the concept map we have created to depict the relationships programs have with other concepts referred to when talking about programs. To begin with, in the following subsection we describe the process underlying the development of the Framework.

3.1 Developing the Framework

The first draft of the document was prepared with an iterative process that involved the entire working group over three months, through regular collaborative weekly virtual meetings. The diversity of researchers in this group was leveraged. The vast expertise covered many different areas of computer science: Theoretical

[6] In the panel discussion *"Media education or computing science? Quo vadis, school teaching?"* which took place at the ISSEP 2022 conference (Vienna, September 27, 2022), Swertz was explicit in manifesting his aversion to mandatory computational thinking in primary education.

[7] The term "nature" is used in the title of the Framework as a colloquial term, and not as a synonym of "ontology".

Computer Science, Algorithms, Software Engineering, Programming Languages, Security and Computer Science Education. Most members of the group have many years of experience in computer science education, both at a research level (covering in particular teaching programming at all levels, and teaching computer science principles to K-12 students) and from a more practical perspective, having participated for years in projects with teachers and schools, and having been involved in teachers' education and curriculum design. A particularly relevant perspective was also brought in by the presence of an epistemologist with expertise in the history of Computer Science, and of programs in particular.

As discussed in Section 2, a variety of models and approaches have been proposed in the literature for characterising programs and programming. Several works describe programs by using "trinities" or more generally tuples, a fact that pinpoints their multifaceted nature:

- In Turner's book [155] programs are defined by the trinity of specification, symbolic program, and physical process.
- Janlert [79, 80] characterises programs in terms of being causative, descriptive and rationally justified.
- In [19] programs are defined as real, abstract, and concrete.
- The PROGRAMme project suggests that a program has three modalities, i.e, different ways of being a program: physical, socio-technical and notational modality.[8]
- Schulte's Block Model [132] considers three dimensions of programs: text surface, program execution, function/purpose.
- According to Jackson [62, 78], requirements, specification, program is a triple connecting the world with the machine in both directions.
- In the definition of free software [57], four essential freedoms are identified that also imply four different ways to interact and deal with a program: using, studying, sharing, improving it.

An analysis and comparison of the tuples was made. It was checked how far they can be mapped on one another and it was concluded that the existing trinities and other tuples only partially overlap. Instead, each of them tends to emphasize some aspects over others, depending on the particular purpose intended when proposing the frameworks. Moreover, each of the various concepts covered by the trinities turned out to be extremely rich and dense in meaning.

The results of the analysis were then contrasted and reviewed in light of the particular purposes inspiring our current work, that is to say, to provide a broad view of the nature of programs for educational purposes, especially at the K–12 levels. The conclusion of the discussion internal to our group was that three components would not be sufficient. First of all, a broadly *accessible* description of the nature of programs was needed and required a finer-grained framework addressed to readers who lack a solid background in computing. Moreover, the research team thought of a framework emphasising not only the program facets that would typically be derived by someone familiar with computing (e.g., code or abstraction) but also those psychological and societal dimensions of human-made artefacts which are especially relevant in educational contexts.

[8]See https://programme.hypotheses.org/

While considering the potential features that characterise programs, a debated point was the role of "Turing universality" and its concrete approximation based on the stored-program concept. According to authoritative scholars of computing history, in particular [65, 104, 142], the *stored-program concept* appears indeed to be regarded "as the defining feature of the modern digital computer, and one that marks the essential transition from calculator to computer" [142] — even though "the dominant understanding of what the stored-program concept is, and of why it is important, has changed considerably over time" [65] and different competent people still provide different answers to the question "what was the significance of the stored program?" [142]. Eventually, however, we opted for not including Turing universality (or the somehow related stored-program concept) as an additional program facet, in that this feature seems to pertain to the structure of the computing device rather than to the program itself.

As a result of the process outlined so far, the research team attempted to answer the question "What is a program?" by identifying six distinct *facets* of programs that made up Part I of the Framework. To complement this, Part II of the Framework focuses on *programming*, i.e. the process of creating and developing programs to address the the question "How does a program come about?" For this part of the project, the starting point was the already cited report [103], which proposes a series of knowledge statements, relating to programming, that every citizen should be aware of; that report was a by-product of another heterogeneously composed working group, installed to define basic knowledge and skills regarding programming, in the context of the revision of Digital Competences Framework 2.1 [30].

In order to explore and elaborate on the content of the report, a preliminary task was set: sketching a diagram that summarized the content of the report, relating it to the six facets of programs already identified. Five different diagrams were proposed and this initiated the discussion about what the fundamental steps are in (small-scale) program development and use, and which programming-related concepts should be included in this research group's framework. Alongside this work, the insights from the literature review were also taken into account. Iterative discussions on these topics led to a brand-new concept map and some derived diagrams, included in Part II of the NoP framework (see Appendix A, from page xxvii), and discussed in Section 3.3.

3.2 Part I - Facets of programs

> *What is a Computer Program?*
> *No doubt virtually all the people who read this sentence believe that they already know what a computer program is. The purpose [...] is to convince you that the question is not so easily answered as you may imagine. Nor is a question without practical value.*
>
> Michael Gemignani (1981) [59]

In this section we consider how *programs* can be characterised, particularly for educational purposes. In the classroom, a student is likely to encounter the concept of a program only implicitly. An educator may be taking it for granted that students will develop a reasonable idea of what a program is by engaging in (simple) programming tasks.

Learning about the nature of an object by gaining experience in how one aspect of it can be manipulated, can help to form a richer understanding of that very same object. However, the learning-by-doing approach has its limitation, especially in the context of programs. First of all, the programs with which students learn to work are far removed from those in common use resulting in a large knowledge gap between actual programs and programs as used in the classroom. This can result in basic misunderstandings when the experiences about programs from the classroom are generalized to the real world examples. Moreover, this gap implies that students are not able to develop a clear vision of where their study might be leading and why particular disciplines within programming (such as using object-oriented programming, or insisting on well-chosen variable names) would pay off in a larger scale system. Secondly, and as we argue here, programs have different facets. Often the examples in the classroom highlight only one or two of these, resulting in a too narrow understanding. Finally, while learning-by-doing is certainly basic from an educational perspective, this focuses only on the how without having any insights into the what. Students should not only know about *how* programs are made, but also about *what* programs are.

We think that the concept of program is richer than it may appear at first, and that it can be better appreciated by taking a *holistic* view, and looking at it from different perspectives. This view also informed our use of term "facets"; instead of assuming one particular perspective on what programs are as a result from one particular tuple, we preferred to use a more neutral approach which speaks of the different facets of a program.

Based on our analysis of the literature, we have identified the following six facets that can be used to characterise programs in terms of three pairs of contrasting features:

notational artefact	**executable entity**
human-made	**tool**
abstract entity	**physical object**

Figure 2 shows a diagrammatic representation of the facets. Some of the facets are likely to appear well-established, even "classical", whereas others may be unexpected. Each of these facets is explained and exemplified in Appendix A, but they are briefly summarised as follows.

- One dominant view of a computer program is that it is "code" — usually referring to source code — and can be seen as a **notational artefact** that follows the rules of a formal language (including natural language in elements such as comments and variable names).
- Balancing this view is the idea that a program is an **executable entity**, running on a physical device that can follow the program's instructions precisely and tirelessly, and will make decisions based on the rules that the programmer decided before the execution began.
- Programs are conceived by people with the intention of satisfying a human need, including induced needs, or the need to express oneself. In this sense, they are **human made**. Commonly, a program is made by teams of people, and the systems developed may consist of many interacting programs. Of course, this is not to say that the final product is entirely hand-made, since automation is an important part of the process, with compilers, interpreters and many other tools being used to support the programming process, but in the end a program is the result of a human decision to create something.
- For many users, a program is a **tool** that they use heavily for both work and leisure. In this context, the user may not particularly understand how the program works, or even what a program is, since it may just appear as an icon on a phone screen that they press to get a weather forecast, or an app that starts running in the background when they open a document for word processing. The program, as a tool, has been crafted to suit the needs of the user, and if this has been done well, the user is able to focus on their task at hand without even thinking about how the tool came to exist.[9]
- A program can be regarded as an **abstract entity** in the sense that it manipulates abstract notions and entities. Computers are often referred to as digital devices because in the end all they do is manipulate binary digits using well defined rules, abstracting real-world entities to a form that can be manipulated by the limited instruction set available to create a program.
- Considering a program as a **physical object** provides us with an insight into its direct presence in the physical world. This is dichotomous to the view that it as an abstract entity, but the instructions in a program must physically exist somewhere (typically in RAM, caches and a CPU while it is running), and the instructions in a program cause physical outcomes when the program is in action.

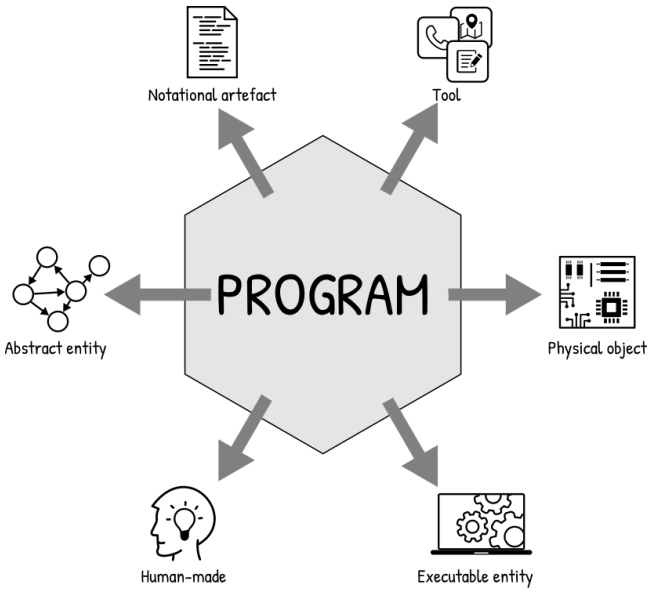


Figure 2: Diagram used to illustrate the six facets

In our perspective, these six identified facets can be seen as different manifestations of programs, but we cannot ignore any one of

[9] Of course, not every program is just a tool unless one has a very broad understanding of "tool". Programs that are created in an art context for instance will not typically be seen as tools (though they might).

them without losing something meaningful about programs. The facets naturally fall into dichotomous pairs, some of which almost appear to contradict each other. These pairs are shown on opposite sides in Figure 2. Incidentally, the tensions notational (script) vs. executable (process) and abstract vs. concrete were already underlying Eden and Turner's attempt to "chart" an ontology of computer programs [47]. In the following sections we focus on these pairs, contrasting these different views, and demonstrating how each provides a valuable viewpoint. Overall, the six facets summarise a range of interesting perspectives on programs.

3.2.1 *Notational Artefact vs. Executable Entity.*

> *In the case of programming languages, the realization of the formulated ideas in the information processing machine is — in a certain sense — guaranteed. This is a fascinating property: writing texts in programming languages cannot only be as creative as poetry, the creations, more than in poetry, belong to the real world as soon as run through the machine.*
>
> Heinz Zemanek (1966) [164]

Regarding this dichotomy, one of the insights comes from elaborating on Eden's ontological analysis [46], which conceptualises programs as the merging of two "facets": *program script* and *program process*. The former sees a program as a piece of text — a *notational artefact*. A program script can then be put on a par with other cultural products. And a cultural object can affect reality, but only indirectly, through the conscious intervention of a human agent who is able to make sense of it. On the other hand, as an executable entity, a program can give rise to a process, i.e, something that occurs, so affecting reality somehow directly, independent of a conscious intervention by a human agent. To put it differently, whoever launches the process (if it is actually a human being) is not required to know what will subsequently happen, and even the intentional user does not need to know anything about the 'how' and the 'why', and does not need to be cognitively involved in the task. Thus, program processes can be *used*, with limited or no awareness of what is going on behind the scenes, whereas program scripts can also be written in order to be *understood*, as a means of expression and communication. A similar distinction between script and process features of programs was made by Zemanek [164]: "Language theory is as well aware as is programming practice that languages have the two aspects of description and prescription: of cognitive vs. instrumental character; of declaration and command; of dealing with states or with actions."

To characterise programs from an educational perspective, i.e. positing the development of logical thinking as a major goal, the expressive side of programs definitely plays a prominent role. In a nutshell, quoting Sussman and colleagues [1, 141], "a computer language is not just a way of getting a computer to perform operations but rather it is a novel formal medium for expressing ideas about methodology. Thus programs must be written for people to read, and only incidentally for machines to execute." Or, in Green's words [61], we "should think of [programs] also as being for communication from ourselves to others, and as vehicles for expressing our own thoughts to ourselves." In particular, Green's quote makes the important role of introspection explicit.

Like natural languages, a programming language should then be endowed with expressive power, but unlike natural languages, what can be expressed with it must be intrinsically accurate, rigorous, and unambiguous. In Minsky's words [110], "If we view a program as a process, we can remember that our most powerful process-describing tools are programs themselves, and they are inherently unambiguous." This point was also emphasised by Zemanek [164] from a semiotic perspective: "The language is the carrier and the implementation of ideas; since it is very hard to handle ideas in an abstract form, the language is an important instrument for the expression, refinement and precision of ideas. So a programming language is also a means of communication between a human being and himself."

We are used to think of the program notation as a device to compose texts, but in fact different types of representational means can have this function as well. For instance, the scholars participating in the multi-voice discussion summarised in [156] reflect about commonalities and differences among three programming environments characterised by non-standard types of notation, with the potential of reifying "objects so that the result of command execution is visible as the position, size, rotation, and other visible state of the object changes" — what, in particular, encourages "*tinkering* as a style of interaction." (The peculiarities of each such environment, more specifically *Alice*, *Greenfoot* and *Scratch*, are addressed in more detail in the following papers of the same TOCE issue [32, 92, 107].) In general, however, "[d]espite decades of study, what makes a computer programming language easy to use for people of all skill levels remains elusive" [140].

Regarding the process side of a program, again from an educational standpoint, a significant feature is that it should be *amenable to experimenting* with it, that is to say, it is somehow "testable" via execution by some unaware processing agent — more specifically, unaware of the program's purpose. The implied agent is of course reminiscent of Wing's (et al.) *information-processing agent* [162] and is not necessarily a computer or some other type of computing artefact, but in principle may also be a human *passive* instruction-follower. In this respect, it is likely that the presumed "unawareness" of the processing agent needs some further clarification, since the borderline between what pertains to the processing agent and what pertains to the programmer may not be so clear-cut. The agent is indeed expected to know how to carry out a range of basic atomic and structured actions the programmer relies on, which depends on some (often implicit) context. In addition, very basic operations under the responsibility of the agent are usually understood "from scratch" in *functional* rather than merely *mechanical* terms, i.e. as if the agent was aware of how to achieve particular (sub-)goals — this is the case, for instance, of the arithmetic and relational operators. To sum up: (i) when dealing with a processing agent, it is necessary to make the context explicit in every detail; (ii) most commonly, such an agent is, in its turn, a program process resulting from a programming task at a different level.

A final observation, from a linguistics point of view, comes from Tanaka-Ishii's essay on the semiotics of programming languages [145]: "Computer languages are the only existing large-scale sign systems with an explicit, fully characterised interpreter external to the human interpretive system." According to Tanaka-Ishii's

analysis, what makes the nature of programming languages different from that of natural languages is that the meaning of program *signs* (e.g. variable and function names) is articulated and fully determined by their use within the program itself, and not by references to exterior objects, namely the mental models of objects in the problem domain, as the signs could be understood by the programmer. This perspective also affects the way we can conceive the relationships between more conceptual semantic models from the one hand, in particular *denotational semantics*, and *operational semantics* models on the other. Although the denotational semantics of a programming language, once it has been formalised, is able to fully explain the meaning of programs written in that language, the very structure of its definition is actually driven by an operational model [155] of how the program behaves.

3.2.2 Human-Made vs. Tool.

> *[T]he relationships between function and structure are central to our grasp of these concepts. Getting from function to structure is a creative activity; it is the crucial design stage of engineering. [...] It is also not possible to extract function from structure [...]. While by testing and experimentation we may be able to figure out what a device actually does, this may not be what it was intended to do [...]. A functional description must provide a black-box description that expresses what it is intended to do. While the structure determines what it actually does, the function is supposed to tell us what it ought to do. That is, the notion of function, while having a propositional content, is intentional in nature.*
>
> Raymond Turner (2018) [155]

The amenability of programs to be subjected to experiments brings the program's intended *purpose* into play. As remarked by Pair [116], sometimes "the calculation is the end in itself, for example, if it controls cartoons, or a game, or (more rarely) a robot; i.e. one is interested in all or some of the stages through which the machine, or a mechanism it controls, passes." A noticeable example in the educational context is programming as *storytelling*, e.g. [86, 150], where the plan, the order of events is all that matters: program and specification can hardly be distinguished. More often, however, what we are interested in is not the way processing goes on, but its final result, which can be identified by *interpreting* the final state of a computation. And in this case the problem of *justification* arises: Does the result actually achieve the intended purpose? Why? In Janlert's opinion [79, 80], in particular, justification should be a necessary condition to regard something as a program: beyond being *causative* (causing a process) and *descriptive* (describing that process), a program is required to be *justified*, i.e. to have "a rational explanation in terms of the goal of the process."

From this perspective, a justification explains the logical connections between program structure and program purpose. The dichotomy of *structure* vs. *function* can be found at the core of the philosophy of *technological artefacts* [94, 155] and inspires Schulte's Block Model framework [132] and its application to program comprehension tasks [77]. In this respect, programs — conceivably together with the "hard" physical devices that support them [155] and give rise to the related processes — represent instances of *human-made* technology, where the task of the designer is to creatively devise suitable relationships between structure and function. Then, ideally, program design, or *synthesis*, proceeds from function to structure. On the other hand, program *analysis*, i.e. proceeding the other way round from structure to function, is also very important from an educational standpoint (program comprehension), and program development is hardy a linear progression in practice anyway. In both cases — of synthesis and analysis — the program structure is understood based on a mental model of the computation and the underlying *notional machine* [43, 54, 139], whereas the program function (or purpose) bridges to a model of the application domain.

In light of the *intended* function (the purpose the designer had in mind), a program can be seen as a *tool* from the user's perspective. The role of software systems as providing tools is at the core, for instance, of Brooks' Jr. [20] view of the computer scientist as a "toolmaker," who "succeeds as, and only as, the users of his tool succeed with his aid." Although mere use of programs is not, per se, a major aim in computer science education, it could nevertheless help "to make investigation of structure meaningful [...] to the learner," as pointed out by Schulte in his proposal of a "duality reconstruction" process to connect *technology* "with individual and social experiences and practices" [133]. In addition, use plays a significant role within such pedagogical approaches as "Use-Modify-Create" (UMC) [99], "Trial-Use-Configure-Create" [134], or "Predict-Run-Investigate-Modify-Make" (PRIMM) [136]. Incidentally, in addition to the intended function, we may observe the emergence of unanticipated practices that can be ascribed to the users' creativity, psychology and social interactions.

Besides the conceptual aspects summarised above, devising, developing and using — or perhaps misusing — technologies presuppose either explicit or implicit values, and have cultural, ethical and societal implications that may be intentional or even unintentional. As seen in section 2.5, several such concerns have prompted critical reflection about sustainable ways to include computing in K–12 curricula. One point on which all educators can be expected to agree with, however, is that the values embodied in the technological artefacts as well as the actual or potential impact on people's lives and relationships should be made as visible as possible to the students.

3.2.3 Abstract Entity vs. Physical Object.

> *As object-oriented programmers, our language says that things as various as shopping carts, chat rooms, and network sockets exist, in some sense, in our computational processes. Have we, in 60 years, come any closer to saying what there is in a computational process? Do our powers of abstraction as programmers have any effect on what there is?*
>
> Timothy Colburn (2006) [28]

Although apparently antithetical to each other, concrete and abstract traits can coexist when we think about programs. On the one hand, we usually make sense of the behaviour of a program by envisaging a world of abstractions that somehow come alive in our mind, thus actually dealing with an *abstract entity*. On the other, it is also a concrete *physical object* as soon as we code, manipulate and run it by means of a computing device. Because of this sort of ambivalence, Hailperin et al. define computer science as "*the discipline of concrete abstractions*" [67].

Programs as *abstract objects* can be put on a par with mathematical abstractions — see Eden [46]. Programs as *concrete objects* can be

approached both from an engineering perspective, focusing on the design of (concrete) technological artefacts, as well as from a scientific perspective, seeing programs as (concrete) objects to explore — see again Eden [46]. These three perspectives — mathematical, engineering and scientific — lead to different conceptualisations of programs in terms of ontology, methodology and epistemology. However, all three are involved to some extent in any standard program development process: the abstract view is prominent when the text of the program, including its *documentation*, is meant to explain to others how it solves the problem at hand; the engineering perspective is taken in the *design* and *refactoring* stages; finally, and probably for much of the development time, an experimental approach underlies the *testing* and *debugging* stages, when the program behaviour turns out to be partially unknown, which makes the actual program (as opposed to the ideal one) an object to explore. Experimenting with computing artefacts, in a scientific sense, has also been proposed by Schulte [134] as a suitable pedagogical means to motivate learners "to uncover structure behind function."

3.2.4 *Less "Traditional" Types of Programs from the multifaceted perspective.*

> *There are now other ways of producing programs, however, that do not seem to require that anyone has had a clear comprehension of how the problem is solved, in the sense of having a subjective method. Two examples of such techniques [...] are: training an artificial neural network to do the task; using an evolutionary algorithm to evolve a program.*
>
> Lars-Erik Janlert (2008) [80]

Janlert [80] distinguishes between "traditional" and "dark" programs. The former result from formalising *subjective methods*, i.e. procedures with a transparent rationale, that in principle could be carried out mindfully by the designer him/her-self. The latter are instead produced by processes with a very different nature, such as training *artificial neural networks* or *evolutionary techniques*.

Another way to explain the peculiar nature of dark programming forms refers to the distinction between intensional vs. extensional definitions in mathematics, e.g. [56]. In more traditional forms of programming, we are primarily concerned with the *intension* of the set of pairs defining the input/output relationship — in particular by characterising a suitable algorithm: "An algorithm which computes a function is a function definition [...]. In this case, the extension is the eventual output, whereas the actual computation is the intension" [56]. In "dark programming", on the other hand, we are concerned directly with the *extension*, namely, (a sample of) the pairs that belong to the input/output relation.

How do these two types of programs relate to the different program facets listed above? As technological artefacts being understood from a structure/function perspective, traditional and dark programs have essentially a similar nature. Likewise, in both cases the concrete physical instantiations of programs, on the one hand, as well as the processes they give rise to, on the other, can be considered alike in nature.

However, a major distinction can be drawn regarding the abstract facet of programs. Abstractness pertains to and makes sense only as a construction of the human mind. Now, in any nontrivial sense what is commonly meant by an abstract understanding of a traditional program cannot be equated to an abstract understanding of a dark program. Abstraction for a traditional program (program comprehension) presupposes, at least in principle, complete knowledge and full mastery of the logic by which that program achieves the intended goals, regardless of running it on a physical device. The approach with dark programs, on the other hand, is usually more empirical, and develops from incomplete or somehow more fragmentary strategic knowledge before executing them.

These different types of abstraction also have implications for our understanding of the nature of program scripts. Unlike the case of dark programs, the script of a traditional program can also (if not mainly) be aimed at conveying and communicating the underlying (abstract) ideas — at least, unless it is just treated as a mere "bunch of data," as in Van der Poel's scenario reported in [112]: "a program is a piece of information only when it is executed. Before it's really executed as a program in the machine it is handled, carried to the machine in the form of a stack of punch cards, or it is transcribed, whatever is the case, and in all these stages, it is handled not as a program but just as a bunch of data." And a "bunch of data" nature would in fact apply equally well to dark program scripts.

Because of the significant distinctions outlined above with regard to the abstract view of programs and the role of program scripts, as well as in order to avoid introducing too many concepts that may potentially confuse the readers, we eventually decided to devise a framework focused on "traditional" programs. In the latter respect, for instance, students' difficulty to appreciate "the conceptual difference between training a neural net and the finished product, a trained neural network," has been observed by [75].

3.3 Part II - How programs are created

> *At its core, programming is about taking a problem defined in the problem domain and building a solution using the tools of the solution domain.*
>
> Alexandron et al. (2014) [4]

Part II of the framework complements the discussion on programs' facets by depicting, through a concept map, the relationships of programs with other concepts that are usually referred to when talking about programs. The map is shown in Figure 3.

As pointed out by Alexandron and colleagues [4], meaningful programs must be understood in connection with some extrinsic (to the program) context that imparts the meaning. The significance of the relationships between the machine and the "outside world" is emphasised, in particular, in the software engineering milieu: see e.g. Jackson [78], who argues how basic it is to try and find a balance between the competing demands of these two contexts.

The definitions of "program" presented in Section 2.1 can be reviewed in light of the concept map and the related description. As in Hopper's definition [70], a program implements an algorithm, which provides a *method to solve a problem*; at the same time a program formalises an algorithm in a *programming language* understandable by the machine, as in Knuth's definition [87]. Dijkstra's focus on *symbol manipulation*, made concrete by the presence of a computer [40], can be found in the concept map's relationship between the computing system and the *digitally encoded data* that it *processes*. Finally, the implementation of an algorithm as a program must comply to specific computing system's *constraints*, as those mentioned in Denning's definition [37].

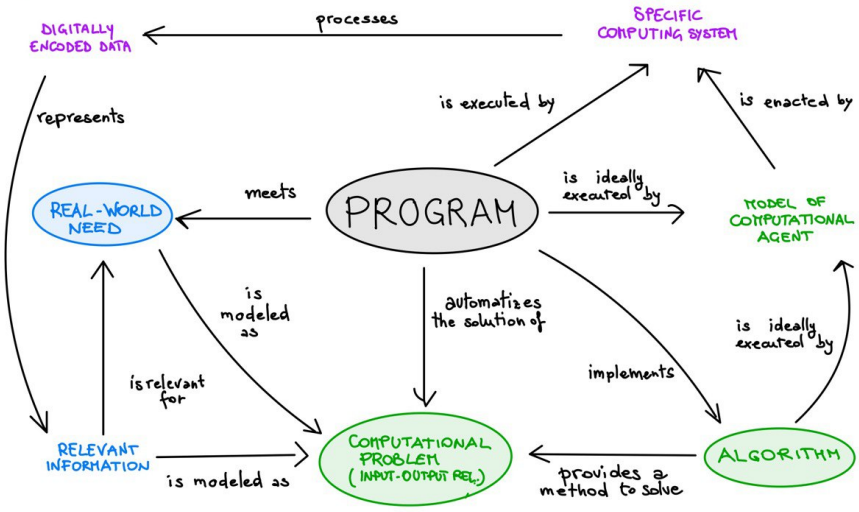


Figure 3: Concept map showing the relationship between programs and other relevant concepts.

Some precursors of the proposed concept map can also be found in Eden and Turner's program abstractions taxonomy [47], where they tie through "concretisation" processes the *Programs* category to other related categories, namely: *Program Scripts* – subdivided in their turn into *Source Code* and *Machine Code*; *Program Processes* – their dynamic and then temporal counterpart, concerning the program execution; *Metaprograms* – abstract program descriptions such as algorithms, automata, design specifications; *Hardware* – physical computing machines.

Even though most of the concepts in the map are already mentioned or implied in Part I as well, the goal of Part II is to better clarify the relationships as well as the distinctions between them. Some of the components represented in the concept map can also be seen in connection with the abstraction hierarchy introduced by Perrenet & Kaasenbrood [123], who identified four levels of program comprehension:

1. *execution level*, focusing on a specific run of the program on a concrete specific machine;
2. *program level*, seeing the program as a process, described in a specific programming language;
3. *object level*, understanding the algorithm described by the program, independent of any specific programming language;
4. *problem level*, where the algorithm can also be viewed as a black box solution to some given problem.

Three of the concepts in the map, namely, the computational problem (problem level), the idealised computational agent (a basis for the program level), and the algorithm (object level), are clearly connected to the abstract facet of programs. The algorithm, in particular, represents a central notion in computer science and it is very close to the concept of program, in that they both describe methods to automatise the solution of a problem. As we see in Perrenet & Kaasenbrood's hierarchy, what distinguishes an algorithm from a program is the fact that the latter *needs* to be expressed in a formal language, whereas the former is usually described more freely, but still with sufficient accuracy so that one can check its correctness and determine its complexity. However, to allow this formal analysis the algorithms are to be conceived in terms of some machine model (e.g. RAM, PRAM, pointer machine), which is in fact a mathematical description of an idealised computational agent. To wrap up, an algorithm is the *abstract idea* of a *method* to solve a problem, whereas a program is a *formal description* for such a method. This distinction is not clear cut, since for any algorithm to be expressed (either for other people to understand it, or for a computational agent to execute it), some notational medium is needed (see the notational facet of programs), precisely as some notational medium is needed to convey any mathematical idea or, more generally, some linguistic medium is needed to communicate ideas in general. Thus, when we try to define formally the two concepts, the differences between them tend to blur as they engulf each other.[10]

In practice, algorithms are often expressed by a mixed use of natural language and high-level semi-formal languages (*i.e.,* in pseudocode), which are loosely defined if compared to programming languages. This might even introduce some ambiguity in the description of the algorithm — which would instead be removed by using a precisely defined syntax and semantics (or an implemented compiler/interpreter for that matters) — however it offers the benefit of reducing the cognitive load while either explaining or understanding the algorithm, hence enabling to focus on the algorithmic ideas, abstracting from the details of a stricter notational representation. This is particularly important if we consider that algorithms are intended to be understood by people, not by machines. A good example to illustrate this trade-off between detailed accuracy and readability is the formal language that Knuth introduced in order

[10] Simone Martini, personal communication, 2022. See also his presentation held in 2020 available at http://www.cs.unibo.it/~martini/TALKS/Simone-anglais.pdf.

to avoid ambiguities in the description of the algorithms analysed in his seminal work [87, 90]. Interestingly, Knuth himself seems to suggest a move in the opposite direction when elaborating on the idea of literate programming [89].

Beyond their differences as to the appropriate notation, both programs and algorithms do nevertheless imply some underlying "interpreter", that is, a computational agent able to either execute the algorithm ideally or to execute the program physically (the focus of Perrenet & Kaasenbrood's execution level). The identification of such an interpreter might be more or less explicit, but both the programmer and the algorithm designer need to have a consistent operational mental model of it, *i.e.,* they have to assume a predefined set of specific actions that the interpreter is able to carry out. The computational agent underlying the design of an algorithm is idealised, and is conceived at a high level of abstraction, whereas the one implied when writing a program is embodied in a specific computing systems aimed at physically processing digitally encoded data (see the physical and executable entity facets of programs respectively at page ix and xiii in Appendix A). It is worth noting that, for the sake of simplicity, we decided not to distinguish between "source code" and "machine code", since several levels of interpretation or translation can be involved, while the general notion of having a precise description for an executor does not depend on such a layered organisation.

The last concept in the map that relates to the abstract facet of programs is the computational problem, *i.e.,* the information-processing counterpart of a real-world need or endeavour, which is instead connected to seeing programs as tools.

The human-made facet may not seem to be linked to any specific concept in the map. However, it becomes apparent if the concept map is used to illustrate the programming process, *i.e.,* how programs are developed (see p. xviii of the NoP framework in Appendix A). As already discussed in the Introduction, we use the term "programming" to refer to the overall process of creating programs, which encompasses many different activities, such as modelling, algorithm design, program implementation, verification, and validation, all activities that are carried out by humans — possibly with the support of other programs, *e.g.,* editors, compilers or even complex code generators, like the ones that go under the name of "low-code" or "zero-code" platforms.

While in practice the distinction between these activities might be blurred or even intentionally ignored (for example, test-driven development somewhat coalesces specification and verification [10]), we argue that, especially in an educational context, it is worth recognising them separately, because understanding their role in programming is a fundamental step towards making sense of the programs' nature. The order in which we presented them in the framework, however, should not suggest they are to be carried out sequentially; on the contrary, this is in fact the reason why we chose to use a concept map instead of, for example, a flow diagram. In other words, we want to keep an agnostic viewpoint with respect to any software life-cycle or process model.

Software development phases have been studied at length [98]. For the purpose of our framework we considered the five activities mentioned above (modelling, algorithm design, program implementation, verification, and validation). This means broadening the scope with respect to what is usually covered in introductory programming courses (particularly in K-12), which unfortunately are sometimes restricted only to implementing programs (usually, under the term "coding" [103]), or, in the best case, to modelling real-world needs and relevant information, or to designing algorithms to solve problems or accomplish tasks (see also [84]).

The distinction between verification (*i.e.,* testing whether the program does indeed solve the computational problem it is supposed to solve) and validation (*i.e.,* assessing whether the program is appropriate to meet the real-world need from the user perspective) might appear subtle (despite the classical imperative to *"do the right thing, do it right"* [138]), but we deem it very important from the educational point of view and for a user-centered view of computing systems. As a further justification for taking into account this point, it may also be worth observing that the distinction between verification, specifically by formal means, and validation in light of the user's expectations is pointed out in a personal note by Dijkstra [41], who refers to these endeavours in terms of "correctness problem" and "pleasantness problem", respectively.

On the other hand, we left out from the concept map all the complexities arising from considering the (planned or unplanned) evolution of software artefacts and their maintenance. While definitely relevant for the professional programmer (flexibility is in fact one of the key properties of software), these problematics cannot be a priority for general education, since they would introduce a proliferation of technical concepts and their value would be hard to grasp without reference to specific professional practices [24].

4 FEEDBACK ON THE FRAMEWORK

In this section feedback from educators and computer science experts is presented.

A draft of the NoP framework was distributed via several mailing lists reaching people interested in computer science education, including both K-12 teachers and academics. They were asked to complete a questionnaire that was devised by the research team (reported in Appendix B) with questions about the scientific soundness of the document, and its interest, usefulness, and readability for the educational context. The survey used mandatory Likert scale questions and optional open-ended questions. The responses were anonymous and did not contain information that could identify the respondents, the only personal pieces of information being their role as educators and their computer science background expertise.

In total, 31 survey responses were received, with almost all of them coming from academic educators (five were secondary school teachers of computer science, one was a secondary teacher of a different subject, one person listed themselves as both an academic and a K-12 teacher). All the respondents claimed to know about programming, with most identifying themselves as experienced and only four as beginners. All respondents consented that their answers could be analysed and used by the members of WG5 at ITiCSE 2022.

The research team did not expect to attract a large number of respondents as the requirement on the participants was rather onerous (the requirement was to read a 20-page document and then answer a range of questions, many asking respondents to write text). Nevertheless, the responses received provided insightful comments, based on the respondents' expertise and accurate reading of

the framework. 25 out of 31 respondents answered to at least one of the optional open-ended questions; more precisely, 98 unique answers were received and were individually analysed. First they were classified according to their reactions to the framework. These included: strong objections, concerns, suggestions, minor remarks, and appreciations. Those that were classified as "concerns" were then further analyzed identifying recurrent themes, which will be elaborated on further in Section 4.2.

4.1 Summary of answers to Likert's scale questions

Overall, 25 respondents found the idea of characterising programs through the **facets** *useful* and only one disagreed explicitly (five people were neutral); 23 found the **facets** *interesting* (one disagreed, seven neutral), 22 found them *insightful* (five disagreed, four neutral), and 22 found them *easy to understand* (four disagreed, five neutral). The **concept map** was found *clear and easy to understand* by nineteen respondents (five disagreed, seven neutral), *interesting* by 23 respondents (three disagreed, five neutral), and *useful* by 22 (two disagreed, seven neutral). The **description of the processes involved in programming** was found *clear and easy to understand* by 22 respondents (three disagreed, six neutrals). 25 respondents found the **facets** *relevant* from an educational perspective (only one disagreed, five neutral); 25 respondents found the **concept map** *relevant* from an educational perspective (three disagreed, three neutral); the **description of the processes involved in programming** was found *relevant* by 25 respondents (four disagreed, two neutral); 22 found the **examples** given in the Section about how programs are created *relevant* (four disagreed, five neutral). Finally, Figure 4 shows the agreement on the statements: "The description of this facet makes sense", "The impact section of this facet helped me better understand this facet", "The examples given helped me better understand this facet" for each facet.

This feedback came from a small sample, with a strong bias towards academics. Despite this, the feedback is certainly encouraging.

4.2 Summary of answers to open-ended questions

All 98 answers to the open-ended questions were analysed individually. As some of them mentioned different aspects, they were split further into separate comments.

Each comment was classified according to its level of severity. Overall, there were strong objections (by five respondents), concerns (by 17 respondents), suggestions (by 10 respondents), minor remarks (by five respondents), and positive comments (by 10 respondents). In many cases, there were comments with different levels of severity for a single respondent (hence the total is higher than the number of respondents).

The strong objections were about the goal or the scope of the framework and they can be summarized as follows[11]: the content presented in the document is already well-known and it's not clear what the contribution of the NoP framework would be (objections mentioned by three respondents); the scope is ill-defined, as the terms "program" and "programming" mean dramatically different things to different groups of people (objections raised by two respondents, one of them even suggested — with a citation of [157] — that programming is a term to avoid in education); the framework is too computer science oriented (one respondent).

[11] the numbers sum up to six instead of five since one respondent expressed two such objections.

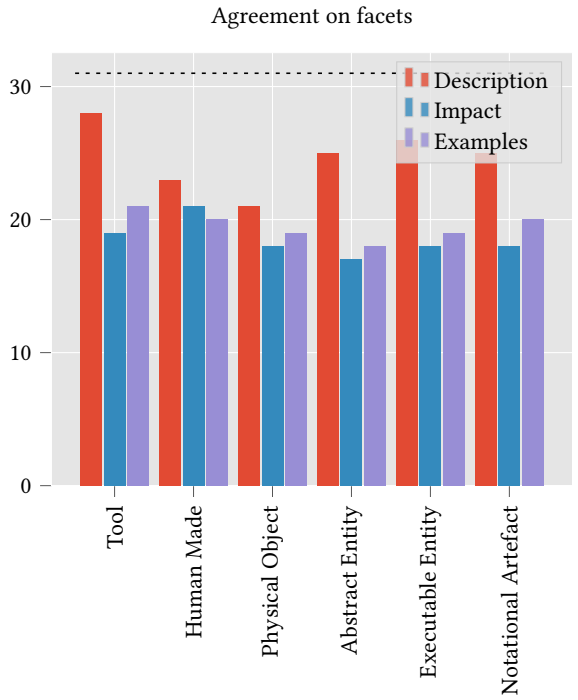


Figure 4: Number of responses with "Agree" or "Strong agree" on Description, Impact, and Examples for facets; the number of respondents is 31, disagreements and neutral responses are not reported.

The comments that raised any concern were analysed in more details. Some concerns were recurrent and regarded:

- *The "physical object" facet* (eight respondents). Some respondents consider the program as separate from its representation; what is physical is the device that runs the programs; some comments suggest a further distinction between the program and its incarnation with measurable physical properties (not only space or mass, but any physical attribute).
- *The "human-made" facet* (four respondents). The fact that sometimes programs are themselves written by other programs is overlooked; one comment was about DNA, a program made by Nature, not humans; and that the role of collaboration in the development of programs is not sufficiently acknowledged.
- *The distinction between "abstract entity" and "notational entity"* (two respondents). It's not clear from the description what is meant with abstraction; the examples for abstract entity involve also notational issues and the distinction between these two facets is not clear.

- *Scope of term "program"* (two respondents). The examples range from simple programs to very complex software systems or services, overlooking their differences, and this may be misleading for non-experts.
- *The evolution and maintenance of software* (two respondents). These aspects are not sufficiently emphasized.

The other comments in this category involved a variety of specific aspects, including:

- human values have a central role and neglecting them can lead to programs that can be used to manipulate users;
- the concept map does not encompass the humanistic perspective;
- in Part I the distinction between algorithms, programs, and software systems is blurred.

Finally, some respondents mentioned aspects that would be worth discussing or at least acknowledging: programs as a computational/mathematical entity; the artistic/aesthetic aspect of programming; the need of unambiguous semantics for a program to be executed; programs as data.

5 DISCUSSION AND FRAMEWORK REVISION

This section discusses the feedback obtained through the survey, and reports on how the framework was revised accordingly.

Overall, the feedback from the respondents is encouraging. Ten respondents commented positively, highlighting and appreciating some particular aspects or parts of the framework that they liked, or the work in its entirety. This is worth noting, as all open-ended questions in the survey asked for constructive criticism of the framework rather than affirmation.

The strongest criticism received objected to the overall purpose of our project, finding its scope too broad or ill defined. As a matter of fact, the very motivation to embark on the enterprise of developing a NoP framework was precisely the fact that the concepts of program and programming are very broad and diversely defined both in the literature and in the educational contexts. However, the criticism led us to realize that the first version of the Framework lacked a clear presentation of its purpose, hence we decided to drastically revise the introduction, change the subtitle, and add an executive summary in order to clarify from the start aims and intended audience. In addition, we stated more explicitly what is covered in the Framework and what is not (e.g., non-traditional programs).

Specific aspects were mentioned that were overlooked, not emphasised enough, or that needed further clarification. Interestingly, most of the concerns raised had already been discussed by the research team during the development of the framework; in several cases, not covering or emphasizing some aspects was an intentional choice, as illustrated in the previous sections (e.g., the decision of leaving out the evolution of software is discussed in Section 3.3). Nevertheless, such comments helped the research team realize that some parts of the exposition did not come across clearly enough or could be enriched, and that some of the choices could be made more explicit in the document.

In what follows next, further detail is provided to describe how the research team addressed the concerns described in Section 4.2.

Concerns about the "physical object" facet. The research team believes that the concreteness of a program is important for practical reasons (the program must be somewhere to be executed, there are no programs in the "clouds" for example) and for grasping its relationship with the machine which executes it. This was self-evident in the past — for example ENIAC's programs were "reified" into visible cable connections — however it is now at risk of being overlooked by many. The description of the "physical entity" facet was revised to clarify this point. Additionally, the following text was moved under the "tool" facet (it was originally inappropriately included under the physical facet as it mentions the physical device):

> A program can turn the same physical device (such as a mobile phone) into a camera, a messaging system, a notepad, a weather prediction system or a game. This is made possible by the fact that the devices are programmable, and this flexibility has led to the ubiquity of programs, since new software can be created to run on existing hardware. Thus, it is not unusual that mobile phone users carry dozens of programs with them, making the phone role just one of the many functionalities of the device.

Concerns about the "human-made" facet. It was made clear, in the introduction of the NoP framework, that the focus is on programs that implement human-designed algorithms, which means that programs produced for instance by evolutionary programming or machine learning techniques are left out.

One respondent objected that DNA (Deoxyribonucleic acid) is a program that is made by nature and not humans. However in our view this remark is not convincing, since, more appropriately, scientists understand the role of DNA in biology as *analogous* to the one that programs have in computing systems. Or, biologists can use DNA or RNA (Ribonucleic acid) to aim at programming a cell, but in that case they act similarly to electronic engineers with voltages, and such programs are still human-made artefacts. This comment led to the following paragraph being inserted under the "human-made" facet:

> One could consider the phenomena of nature itself as systems that process information (think of DNA transcription or plants that use chemicals to transmit information). But, unlike these systems, computer programs are built by humans intentionally to exploit information processing devices.

Concerns about the distinction between "abstract entity" and "notational entity". As remarked by Zemanek [164] (see Section 3.2.1), abstraction implies the use of some notation and then the two facets are necessarily connected in some way. However, it is acknowledged by the research team that many examples illustrating the "abstract entity" (present in the draft version of the framework) relied too much on notational issues, such as:

> Simple programming languages for beginners sometimes have instruction sets like "Forward", "Right" and "Left" to program a sprite or robot to move around. In some languages these might be written as "F", "R"

> and "L" respectively, and in another they may be represented as corresponding arrows. ...
>
> When planning how to implement a program, intermediate notations such as pseudo-code, UML, or even just rough notes, are useful ...

The examples were modified so that they focus mainly on abstraction, and the description was clarified to explain the relation between the two facets.

Concerns about the scope of term "program". It was made explicit in the introduction of the NoP framework that term "program" is used in a broad sense, ranging from simple programs that are studied in computer science classes, to complex software systems or services such as an operating system or a social network. Despite the differences in their sizes, complexity, and user involvement have a very important and clear role (especially considering the development process and the impact on their users), in fact they all share the same *nature*, as described and argued in the multifaceted characterisation presented in the NoP framework.

6 LIMITATIONS

Before moving to the conclusions, here we summarise for the sake of clarity some limitations of our study, most of which have already been discussed above.

Non-traditional programs. Our focus is on programs formalising algorithms fully conceived by the human mind, so that we can explain in detail how the intended goal is being achieved. As discussed in Section 3.2.4, "non-traditional" programs whose implemented *logic* is not directly devised by humans, but results from elusive mechanisms[12] (for example, via evolutionary programming or machine learning techniques) are left out of our present scope.

Maintenance and evolution of software. The concept map presented in Part II of the Framework ignores all the complexities concerning the (planned or unplanned) evolution and maintenance of software artefacts. While definitely relevant for the professional programmer (flexibility is in fact one of the key properties of software), we believe these topics cannot be a priority for *general* education, since they would introduce a proliferation of technical concepts, whose value would be hard to grasp without reference to specific professional practices [24].

Multifaceted characterisation of program nature. The literature review suggests that the nature of programs is multifaceted. The choice of precisely the six particular facets introduced in Section 3.2 was in fact a specific outcome of the discussion within the working group, but we acknowledge that other different choices could have been made. Still, we believe that these six facets are especially meaningful for a number of reasons. First, other kinds of artifacts essentially lack the features of at least one of the discussed facets, making their combination somehow unique to programs. Moreover, each of the six facets reflects relevant aspects that are debated in the literature. Last but not least, identifying these different facets might not be obvious for educators without a background in computer science, whereas we think that this perspective could be inspiring for instructional purposes.

Perspectives about the nature of programs. The literature spans a variety of contributions about the nature of programs from epistemological, philosophical, and historical perspectives. We drew a partial picture of this landscape in Section 2.1; however, we deliberately refrained from delving too deeply into philosophical arguments. Instead, we aimed for an analysis that, on the one hand, relies on the existing literature, and, on the other, targets an audience who, although not specialized in the field, would benefit from a broader understanding of the rich nature of computer programs.

Validation of the Framework. The feedback obtained from the survey respondents is encouraging, but the sample we managed to reach is clearly small and under-representative, particularly in regard to primary and lower-secondary educators. This was mainly due to the time constraints of the working group's project, that did not allow for a broader and more systematic validation of the Framework. However, we are planning to collect additional insights in future follow-up work.

7 CONCLUSIONS

There are many views of what a program is, and by collecting and connecting the different facets that programs display we have uncovered a rich range of angles from which to view this intriguing object that appears in so many circumstances of daily lives. The use of computer programs on digital devices has evolved considerably in the relatively short time-scale that they have existed, and while statements about computer programs from the 1950s are still true, the range of contexts in which programs are used has evolved considerably. In the early days, a computer was a single expensive device owned by some large organisation and was applied to a limited range of purposes, but now individuals own or use multiple devices, and each device is potentially loaded with hundreds of programs. It is therefore not surprising that we can uncover multiple facets that help us to understand what a program is, rather than the narrower characterisations prevailing in the past.

Of course, watertight definitions are elusive, our understanding of technology evolves, and it is difficult to identify clear boundaries to which part of a system is a program. Indeed, very often what appears to be a "program" is a gateway to a profusion of programs; even a simple weather forecast app, when launched, will invoke a variety of network programs communicating with each other, and that in turn collect data produced by a remote program, or group of programs. Hence, our goal has been to open up the view of what a program is, rather than draw a tight boundary around it.

At the current stage of our endeavour, we envisage some possible perspective work around the framework and a few related research directions. To begin with, as most of the responses received were from computer science academics, we plan to collect additional feedback from a larger sample of K-12 teachers spanning all the instruction levels, and possibly from different countries. Since reading the entire document and answering the open-ended questions may be too demanding a task for teachers, instead of using an online survey we think of organizing reading groups where they can discuss together and critique the various aspects of the framework.

[12] Notice that we are referring to the program's logic, and not merely to its code, as machine-produced.

Then, based on teachers' suggestions, we could further refine the NoP framework.

Furthermore, teachers are usually interested in practical, concrete activities to propose in their classes. In this respect, it would be useful to develop, possibly with their cooperation, a repertory of student tasks that highlight the role of each one of the program facets. Finally, the framework could be complemented with annexes that cover more advanced programming topics and/or non-traditional programs.

From a research perspective, it would be interesting to investigate on students' and teachers' views in connection with the six program facets; more specifically, we mean their *spontaneous* views, developed without knowing about the NoP framework. Another research direction may address the question of whether/how students' perception of computer science and programming — and perhaps the perception of teachers involved in PD programs — changes before and after being exposed to the NoP framework.

To sum up, as a result of the developments of our endeavour, we hope that the NoP framework could help educators, curriculum designers and policy makers to have an informed view of programs and programming. This would also enable our students to better understand how they can approach learning to program, with a deeper comprehension of the powerful idea of "program".

ACKNOWLEDGMENTS

This work is supported in part by the Slovenian Research Agency under Grants No.: P2-0359, J2-2504 and J1-2481, and by the French National Research Agency under Grant No.: PROGRAMme, N° ANR-17-CE38-0003-01 (https://anr.fr/Project-ANR-17-CE38-0003).

REFERENCES

[1] Harold Abelson, Gerald J. Sussman, and Julie Sussman. 1984. *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, MA.

[2] Alireza Ahadi and Raymond Lister. 2013. Geek Genes, Prior Knowledge, Stumbling Points and Learning Edge Momentum: Parts of the One Elephant?. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery, New York, NY, USA, 123–128. https://doi.org/10.1145/2493394.2493395

[3] Christopher Alexander et al. 1979. *The timeless way of building*. Vol. 1. Oxford University Press, New York.

[4] Giora Alexandron, Michal Armoni, Michal Gordon, and David Harel. 2014. Scenario-Based Programming, Usability-Oriented Perception. *ACM Trans. Comput. Educ.* 14, 3, Article 21 (oct 2014), 23 pages. https://doi.org/10.1145/2648814

[5] Kashif Amanullah. 2020. *Using elementary patterns to analyse Scratch programs.* Ph. D. Dissertation. University of Canterbury, Canterbury, New Zealand.

[6] Michal Armoni. 2019. COMPUTING IN SCHOOLS – Why Are We Teaching This? Strings and Beyond. *ACM Inroads* 10, 1 (feb 2019), 30–32. https://doi.org/10.1145/3306136

[7] Owen Astrachan and Eugene Wallingford. 1998. Loop patterns. In *Proceedings of the Fifth Pattern Languages of Programs Conference.* Allerton Park, Illinois.

[8] Albert Bandura. 2001. Social Cognitive Theory: An Agentic Perspective. *Annual review of psychology* 52 (2001), 1–26.

[9] Jean Baudrillard. 1981. *Simulacres and simulation.* Editions Galilée, Paris, France. English ed.: Simulacra and Simulation, University of Michigan Press (1994).

[10] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional, Boston, MA.

[11] Tim Bell, Peter Andreae, and Anthony Robins. 2014. A Case Study of the Introduction of Computer Science in NZ Schools. *ACM Trans. Comput. Educ.* 14, 2 (June 2014), 10:1–10:31. https://doi.org/10.1145/2602485 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[12] Tim Bell, Paul Tymann, and Amiram Yehudai. 2018. The Big Ideas in Computer Science for K-12 Curricula. *Bull. EATCS* 124 (2018), 11 pages.

[13] Mordechai Ben-Ari. 2016. In defense of programming. *ACM Inroads* 7, 1 (2016), 44–46.

[14] Joseph Bergin. 2001. Coding at the lowest level: Coding patterns for java beginners. In *EuroPLoP*, Vol. 2001. UVK - Universitaetsverlag Konstanz, Irsee, Germany, 251–286.

[15] Alan F. Blackwell. 2002. First Steps in Programming: A Rationale for Attention Investment Models. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC '02).* IEEE Computer Society, Arlington, Virginia, 2–10.

[16] Alan F. Blackwell. 2002. What is programming?. In *14th workshop of the Psychology of Programming Interest Group* (Brunel University, London, UK). Psychology of Programming Interest Group, UK, 204–218.

[17] Russell Boyatt, Meurig Beynon, and Megan Beynon. 2014. Ghosts of Programming Past, Present and Yet to Come. In *Proceedings of the 25th Annual Workshop of the Psychology of Programming Interest Group – PPIG 2014* (University of Sussex, Brighton, UK), Benedict du Boulay and Judith Good (Eds.). Psychology of Programming Interest Group, UK, 171–182.

[18] Nina Bresnihan, Richard Millwood, Elizabeth Oldham, Glenn Strong, and Diana Wilson. 2015. CA critique of the current trend to implement computing in schools. *Pedagogika* 65, 3 (2015), 292–300. http://userweb.pedf.cuni.cz/wp/pedagogika/ Historical Study.

[19] Andrej Brodnik, Andrew Csizmadia, Gerald Futschek, Lidija Kralj, Violetta Lonati, Peter Micheuz, and Mattia Monga. 2021. Programming for All: Understanding the Nature of Programs. *CoRR* abs/2111.04887 (2021), 25 pages. arXiv:2111.04887

[20] Frederick P. Brooks. 1996. The Computer Scientist as Toolsmith II. *Commun. ACM* 39, 3 (mar 1996), 61–68. https://doi.org/10.1145/227234.227243

[21] Quinn Burke. 2012. The Markings of a New Pencil: Introducing Programming-as-Writing in the Middle School Classroom. *Journal of Media Literacy Education* 4, 2 (April 2012), 121–135.

[22] Quinn Burke, W. Ian O'Byrne, and Yasmin B. Kafai. 2016. Computational Participation: Understanding Coding as an Extension of Literacy Instruction. *Journal of Adolescent & Adult Literacy* 59, 4 (January/February 2016), 371–375. https://doi.org/10.1002/jaal.496

[23] Cathy Burnett. 2016. *The digital age and its implications for learning and teaching in the primary school.* Technical Report. Cambridge Primary Review Trust – CPRT Research Survey 7 (new series), York, UK. www.cprtrust.org.uk

[24] Henrik Bærbak Christensen. 2009. A Story-Telling Approach for a Software Engineering Course Design. In *Proceedings of the 14th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education* (Paris, France) *(ITiCSE '09)*. Association for Computing Machinery, New York, NY, USA, 60–64. https://doi.org/10.1145/1562877.1562901

[25] Wendy Hui Kyong Chun. 2008. On "Sourcery," or Code as Fetish. *Configurations* 16, 3 (2008), 299–324. https://doi.org/10.1353/con.0.0064 Project MUSE.

[26] Michael J. Clancy and Marcia C. Linn. 1999. Patterns and pedagogy. *ACM SIGCSE Bulletin* 31, 1 (1999), 37–42.

[27] Timothy Colburn. 1999. *Philosophy and Computer Science.* Routledge (Taylor & Francis), New York, NY.

[28] Timothy Colburn. 2006. What is Philosophy of Computer Science?. In *Proc. of the European Conference on Computing and Philosophy.* Norwegian University of Science and Technology, Trondheim, Norway, 4 pages.

[29] Timothy Colburn and Gary Shute. 2007. Abstraction in Computer Science. *Minds and Machines* 17, 2 (2007), 169–184.

[30] European Commission, Joint Research Centre, S. Carretero, R. Vuorikari, and Y. Punie. 2018. *DigComp 2.1: the digital competence framework for citizens with eight proficiency levels and examples of use.* Publications Office, Brussels, Belgium. https://doi.org/10.2760/38842

[31] European Commission, Joint Research Centre, R. Vuorikari, S. Kluzer, and Y. Punie. 2022. *DigComp 2.2, The Digital Competence framework for citizens : with new examples of knowledge, skills and attitudes.* Publications Office of the European Union, Brussels, Belgium. https://doi.org/10.2760/115376

[32] Stephen Cooper. 2010. The Design of Alice. *ACM Trans. Comput. Educ.* 10, 4, Article 15 (nov 2010), 16 pages. https://doi.org/10.1145/1868358.1868362

[33] Isabella Corradini, Michael Lodi, and Enrico Nardelli. 2018. An Investigation of Italian Primary School Teachers' View on Coding and Programming. In *Informatics in Schools. Fundamentals of Computer Science and Software Engineering*, Sergei N. Pozdniakov and Valentina Dagienė (Eds.). Springer International Publishing, Cham, 228–243.

[34] Kathryn Cunningham. 2018. The novice programmer needs a plan. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, IEEE Press, Lisbon, Portugal, 269–270.

[35] Paul Curzon, Tim Bell, Jane Waite, , and Mark Dorling. 2019. Computational Thinking. In *The Cambridge Handbook of Computing Education Research*, S. A. Fincher and A. V. Robins (Eds.). Cambridge University Press, Cambridge, Chapter 17, 513–546. https://doi.org/10.1017/9781108654555

[36] Liesbeth De Mol and Maarten Bullynck. 2021. Roots of 'program' Revisited. *Commun. ACM* 64, 4 (mar 2021), 35–37. https://doi.org/10.1145/3419406

[37] Peter J. Denning. 2004. The Field of Programmers Myth. *Commun. ACM* 47, 7 (jul 2004), 15–20. https://doi.org/10.1145/1005817.1005836

[38] Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R Young. 1989. Computing as a discipline.

Computer 22, 2 (1989), 63–70.
[39] Peter J. Denning and Craig H. Martell. 2015. *Great Principles of Computing.* The MIT Press, Cambridge, MA.
[40] Edsger W. Dijkstra. 1988. On the cruelty of really teaching computer science. Personal communication. https://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF
[41] Edsger W. Dijkstra. 1989. In reply to comments. (1989). http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1058.PDF circulated privately.
[42] Gill Dowek et al. 2012. *Informatique et sciences du nmérique.* Éditions Eyrolles, Paris, France.
[43] Benedict du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.
[44] Benedict du Boulay, Tim O'Shea, and John Monk. 1981. The black box inside the glass box: presenting computing concepts to novices. *International Journal of man-machine studies* 14, 3 (1981), 237–249.
[45] Tomi Dufva and Mikko Dufva. 2016. Metaphors of code – Structuring and broadening the discussion on teaching children to code. *Thinking Skills and Creativity* 22 (2016), 97–110. https://doi.org/10.1016/j.tsc.2016.09.004
[46] Amnon H. Eden. 2007. Three Paradigms of Computer Science. *Minds Mach.* 17, 2 (jul 2007), 135–167. https://doi.org/10.1007/s11023-007-9060-8
[47] Amnon H. Eden and Raymond Turner. 2007. Problems in the Ontology of Computer Programs. *Applied Ontology* 2, 1 (2007), 13–36.
[48] Nathan Ensmenger. 2010. *The Computer Boys Take Over. Computers, Programmers, and the Politics of Technical Expertise.* MIT Press, Cambridge MA.
[49] Fabian Fagerholm and Arto Hellas. 2020. On the Differences in Time That Students Take to Write Solutions to Programming Problems. In *2020 IEEE Frontiers in Education Conference (FIE).* IEEE, IEEE Press, Lincoln, NE, USA, 1–9.
[50] Roisin Faherty, Keith Quille, Rebecca Vivian, Monica M. McGill, Brett A. Becker, and Karen Nolan. 2021. Comparing Programming Self-Esteem of Upper Secondary School Teachers to CS1 Students. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1* (Virtual Event, Germany) *(ITiCSE '21).* Association for Computing Machinery, New York, NY, USA, 554–560. https://doi.org/10.1145/3430665.3456372
[51] Katrina Falkner, Sue Sentance, Rebecca Vivian, Sarah Barksdale, Leonard Busuttil, Elizabeth Cole, Christine Liebe, Francesco Maiorana, Monica M. McGill, and Keith Quille. 2019. An International Comparison of K-12 Computer Science Education Intended and Enacted Curricula. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '19).* Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3364510.3364517
[52] Katrina Falkner and Rebecca Vivian. 2015. A review of Computer Science resources for learning and teaching with K-12 computing curricula: an Australian case study. *Computer Science Education* 25, 4 (2015), 390–429. https://doi.org/10.1080/08993408.2016.1140410 Publisher: Routledge _eprint: https://doi.org/10.1080/08993408.2016.1140410.
[53] James H. Fetzer. 1988. Program Verification: The Very Idea. *Commun. ACM* 31, 9 (sep 1988), 1048–1063. https://doi.org/10.1145/48529.48530
[54] Sally Fincher, Johan Jeuring, Craig S. Miller, Peter Donaldson, Benedict du Boulay, Matthias Hauswirth, Arto Hellas, Felienne Hermans, Colleen Lewis, Andreas Mühling, Janice L. Pearce, and Andrew Petersen. 2020. Notional Machines in Computing Education: The Education of Attention. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE-WGR '20).* Association for Computing Machinery, New York, NY, USA, 21–50. https://doi.org/10.1145/3437800.3439202
[55] Bjarke Vognstrup Fog and Clemens Nylandsted Klokmose. 2019. Mapping the Landscape of Literate Computing. In *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group – PPIG 2019*, Mariana Marasoiu, Luke Church, and Lindsay Marshall (Eds.). Psychology of Programming Interest Group, Newcastle University, UK, 10 pages.
[56] Daniel Fredholm. 1995. Intensional aspects of function definitions. *Theoretical Computer Science* 152, 1 (1995), 1–66. https://doi.org/10.1016/0304-3975(94)00268-9
[57] Free Software Foundation. 2021. https://www.gnu.org/philosophy/free-sw.html.en. v1.69.
[58] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, Upper Saddle River, New Jersey.
[59] Michael Gemignani. 1981. What is a Computer Program? *The American Mathematical Monthly* 88, 3 (1981), 185–188. http://www.jstor.org/stable/2320464
[60] Chris Granger. 2014. Towards A Better Programming. Posted by the author. http://www.chris-granger.com/2014/03/27/toward-a-better-programming/ Retrieved: May 2022.
[61] Thomas R. G. Green. 1990. The Nature of Programming. In *Psychology of Programming*, J.-M. Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore (Eds.). Academic Press, London, Chapter 1.2, 21–44. https://doi.org/10.1016/B978-0-12-350772-3.50007-0
[62] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. 2000. A reference model for requirements and specifications. *IEEE Software* 17, 3 (2000), 37–43. https://doi.org/10.1109/52.896248
[63] Mark Guzdial. 2019. Technical perspective: Is there a geek gene? *Commun. ACM* 63, 1 (2019), 90–90.
[64] Mark Guzdial and Barbara Ericson. 2007. *Introduction to Computing and Programming in Java: A Multimedia Approach.* Prentice-Hall, Upper Saddle River, NJ.
[65] Thomas Haigh, Mark Priestley, and Crispin Rope. 2014. Reconsidering the Stored-Program Concept. *IEEE Annals of the History of Computing* 36, 1 (2014), 4–17. https://doi.org/10.1109/MAHC.2013.56
[66] Thomas Haigh, Mark Priestley, and Crispin Rope. 2016. *ENIAC in action: making and remaking the modern computer.* The MIT Press, Cambridge, MA.
[67] Max Hailperin, Barbara Kaiser, and Karl Knight. 1999. *Concrete Abstractions.* Brooks/Cole, Pacific Grove, CA.
[68] Fredrik Heintz and Linda Mannila. 2018. Computational Thinking for All: An Experience Report on Scaling up Teaching Computational Thinking to All Students in a Major City in Sweden. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18).* Association for Computing Machinery, New York, NY, USA, 137–142. https://doi.org/10.1145/3159450.3159586 event-place: Baltimore, Maryland, USA.
[69] Jason Hong. 2022. Modern Tech Can't Shield Your Secret Identity. *Commun. ACM* 65, 5 (apr 2022), 24–25. https://doi.org/10.1145/3524013
[70] Grace M. Hopper, chairman. 1954. ACM first glossary of programming terminology. Report to the ACM from the Committee on Nomenclature.
[71] Juraj Hromkovič. 2006. Contributing to General Education by Teaching Informatics. In *Proceedings of the International Conference on Informatics in Secondary Schools – Evolution and Perspectives (ISSEP)* (Vilnius, Lithuania) *(LNCS, Vol. 4226)*, Roland T. Mittermeir (Ed.). Springer, Berlin / Heidelberg, 25–37.
[72] Juraj Hromkovič. 2009. *Algorithmic Adventures: From Knowledge to Magic.* Springer, Berlin/Heidelberg, Germany.
[73] Peter Hubwieser, Michal Armoni, Michail N. Giannakos, and Roland T. Mittermeir. 2014. Perspectives and Visions of Computer Science Education in Primary and Secondary (K-12) Schools. *ACM Trans. Comput. Educ.* 14, 2, Article 7 (jun 2014), 9 pages. https://doi.org/10.1145/2602482
[74] Ioanna Iacovides and Thomas R.G. Green. 2021. Neither Grasshopper nor Ant: learning from DIY coding and from gaming. In *Proceedings of the 32nd Annual Workshop of the Psychology of Programming Interest Group – PPIG 2021*, UK) Thomas Green (York and Clayton Lewis (Eds.). Psychology of Programming Interest Group, virtual, 6 pages.
[75] Susan P. Imberman. 2004. An Intelligent Agent Approach for Teaching Neural Networks Using LEGO® Handy Board Robots. *J. Educ. Resour. Comput.* 4, 3 (sep 2004), 4–es. https://doi.org/10.1145/1083310.1083312
[76] Nurbay Irmak. 2012. Software is an Abstract Artifact. *Grazer Philosophische Studien* 86, 1 (2012), 55–72.
[77] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, and et al. 2019. Fostering Program Comprehension in Novice Programmers - Learning Activities and Learning Trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education* (Aberdeen, Scotland Uk) *(ITiCSE-WGR '19).* Association for Computing Machinery, New York, NY, USA, 27–52. https://doi.org/10.1145/3344429.3372501
[78] Michael Jackson. 1995. The World and the Machine. In *Proceedings of the 17th International Conference on Software Engineering* (Seattle, Washington, USA) *(ICSE '95).* Association for Computing Machinery, New York, NY, USA, 283–292. https://doi.org/10.1145/225014.225041
[79] Lars-Erik Janlert. 2006. The program is the solution — what is the problem? European Conference on Computing and Philosophy.
[80] Lars-Erik Janlert. 2008. Dark programming and the case for the rationality of programs. *Journal of Applied Logic* 6, 4 (2008), 545– 552. https://doi.org/10.1016/j.jal.2008.09.003 The Philosophy of Computer Science.
[81] Bo Ju, Olivia Ravenscroft, Evelyn Flores, Denise Nacu, Sheena Erete, and Nichole Pinkard. 2020. Understanding Parents' Perceived Barriers to Engaging Their Children in Out-of-School Computer Science Programs. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education.* Association for Computing Machinery, New York, NY, USA, 1272. https://doi.org/10.1145/3328778.3372663
[82] Yasmin B. Kafai, Quinn Burke, and EBSCOHost. 2014. *Connected code: why children need to learn programming.* The MIT Press, Cambridge, Massachusetts.
[83] Alan Kay. 1984. Computer software. *Scientific American* 251 (1984), 53–59.
[84] Therese Keane and Andrew Fluck. 2022. *Teaching Coding in K-12 Schools – Research and Application.* Springer Nature, Cham, Switzerland.
[85] Caitlin Kelleher and Randy Pausch. 2005. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv.* 37 (June 2005), 83–137. Issue 2. https://doi.org/10.1145/1089733.1089734
[86] Caitlin Kelleher and Randy Pausch. 2007. Using storytelling to motivate programming. *Commun. ACM* 50, 7 (2007), 58–64.
[87] Donald E. Knuth. 1968. *The Art of Computer Programming (1st Edition).* Vol. 1. Addison-Wesley, Boston, MA.

[88] Donald E. Knuth. 1974. Computer Science and its Relation to Mathematics. *The American Mathematical Monthly* 81, 4 (1974), 323–343. https://doi.org/10.1080/00029890.1974.11993556 arXiv:https://doi.org/10.1080/00029890.1974.11993556

[89] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (01 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[90] Donald E. Knuth. 2005. *The Art of Computer Programming, Volume 1, Fascicle 1: MMIX–A RISC Computer for the New Millennium.* Addison-Wesley Professional, Boston, MA.

[91] Amy J. Ko. 2016. What is a Programming Language, Really?. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools* (Amsterdam, Netherlands) *(PLATEAU 2016)*. Association for Computing Machinery, New York, NY, USA, 32–33. https://doi.org/10.1145/3001878.3001880

[92] Michael Kölling. 2010. The Greenfoot Programming Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 14 (nov 2010), 21 pages. https://doi.org/10.1145/1868358.1868361

[93] Shriram Krishnamurthi and Kathi Fisler. 2019. Programming Paradigms and Beyond. In *The Cambridge Handbook of Computing Education Research*, S. A. Fincher and A. V. Robins (Eds.). Cambridge University Press, Cambridge, Chapter 13, 377–413. https://doi.org/10.1017/9781108654555

[94] Peter Kroes. 2012. *Technical Artefacts: Creations of Mind and Matter – A Philosophy of Engineering Design.* Springer, Dordrecht, Heidelberg, New York, London. https://doi.org/10.1007/978-94-007-3940-6

[95] Logan Kugler. 2022. Addressing Labor Shortages with Automation. *Commun. ACM* 65, 6 (may 2022), 21–23. https://doi.org/10.1145/3530687

[96] Pascal Lando, Anne Lapujade, Gilles Kassel, and Frédéric Fürst. 2007. Towards a General Ontology of Computer Programs. In *ICSOFT 2007, Proceedings of the Second International Conference on Software and Data Technologies, Volume PL/DPS/KE/MUSE*, Joaquim Filipe, Boris Shishkov, and Markus Helfert (Eds.). INSTICC Press, Barcelona, Spain, 163–170.

[97] Brett J. L. Landry and M. Scott Koger. 2006. Dispelling 10 Common Disaster Recovery Myths: Lessons Learned from Hurricane Katrina and Other Disasters. *J. Educ. Resour. Comput.* 6, 4 (dec 2006), 6–es. https://doi.org/10.1145/1248453.1248459

[98] Craig Larman and Victor R. Basili. 2003. Iterative and incremental developments. a brief history. *Computer* 36, 6 (2003), 47–56. https://doi.org/10.1109/MC.2003.1204375

[99] Irene Lee, Fred Martin, Jill Denner, Bob Coulter, Walter Allan, Jeri Erickson, Joyce Malyn-Smith, and Linda Werner. 2011. Computational Thinking for Youth in Practice. *ACM Inroads* 2, 1 (Feb. 2011), 32–37. https://doi.org/10.1145/1929887.1929902 Place: New York, NY, USA Publisher: Association for Computing Machinery.

[100] Meir M. Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076. https://doi.org/10.1109/PROC.1980.11805

[101] Raymond Lister. 2011. Computing education research geek genes and bimodal grades. *ACM Inroads* 1, 3 (2011), 16–17.

[102] Michael Lodi. 2017. Growth Mindset in Computational Thinking teaching and teacher training. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (Tacoma, Washington, USA) *(ICER '17)*. Association for Computing Machinery, New York, NY, USA, 281–282. https://doi.org/10.1145/3105726.3105736

[103] Violetta Lonati, Dario Malchiodi, Mattia Monga, and Anna Morpurgo. 2015. Is coding the way to go?. In *8th International Conference on Informatics in Schools: Situation, Evolution, and Perspective* (Ljubljana, Slovenia) *(LNCS, Vol. 9378)*, Andrej Brodnik and Jan Vahrenhold (Eds.). Springer International Publishing, Switzerland, 165–174. https://doi.org/10.1007/978-3-319-25396-1_15

[104] Michael S. Mahoney. 2005. The histories of computing(s). *Interdisciplinary Science Reviews* 30, 2 (2005), 119–135. https://doi.org/10.1179/030801805X25927

[105] Michael S. Mahoney. 2008. What Makes the History of Software Hard. *IEEE Annals of the History of Computing* 30, 3 (2008), 8–18. https://doi.org/10.1109/MAHC.2008.55

[106] Lauri Malmi, Judy Sheard, Päivi Kinnunen, Simon, and Jane Sinclair. 2020. Theories and Models of Emotions, Attitudes, and Self-Efficacy in the Context of Programming Education. In *Proceedings of the 2020 ACM Conference on International Computing Education Research (ICER '20)*. Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/3372782.3406279 event-place: Virtual Event, New Zealand.

[107] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *ACM Trans. Comput. Educ.* 10, 4, Article 16 (nov 2010), 15 pages. https://doi.org/10.1145/1868358.1868363

[108] Pekka Mertala. 2021. The pedagogy of multiliteracies as a code breaker: A suggestion for a transversal approach to computing education in basic education. *British Journal of Educational Technology (BERA)* 52, 6 (2021), 2227–2241. https://doi.org/10.1111/bjet.13125

[109] Eric M. Meyers. 2019. Guest editorial – Learning to code, coding to learn: youth and computational thinking. *Information and Learning Sciences* 120, 5/6 (11 2019), 254–265. https://doi.org/10.1108/ILS-05-2019-139

[110] Marvin Minsky. 1970. Form and Content in Computer Science. In *ACM Turing Award Lectures*. Association for Computing Machinery, New York, NY, USA, 197–215. https://doi.org/10.1145/1283920.1283924

[111] Sukanya Kannan Moudgalya, Aman Yadav, Philip Sands, Sara Vogel, and Mike Zamansky. 2021. Teacher Views on Computational Thinking as a Pathway to Computer Science. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. Association for Computing Machinery, New York, NY, USA, 262–268. https://doi.org/10.1145/3430665.3456334

[112] Peter Naur and Brian Randell (Eds.). 1969. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968.* Scientific Affairs Division – NATO, Brussels, Belgium.

[113] Tom Neutens and Francis Wyffels. 2018. Bringing Computer Science Education to Secondary School: A Teacher First Approach. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 840–845. https://doi.org/10.1145/3159450.3159568 event-place: Baltimore, Maryland, USA.

[114] James Noble and Robert Biddle. 2002. Notes on Postmodern Programming. In *Proceedings of the Onward Track at OOPSLA '02* (Seattle, USA), Richard Gabriel (Ed.). Association for Computing Machinery, New York, NY, USA, 49–71. http://www.dreamsongs.org/ the ACM conference on Object-Oriented Programming, Systems, Languages and Applications.

[115] James Noble and Robert Biddle. 2004. Notes on Notes on Postmodern Programming. *SIGPLAN Not.* 39, 12 (Dec 2004), 40–56. https://doi.org/10.1145/1052883.1052890

[116] C. Pair. 1990. Programming, Programming Languages and Programming Methods. In *Psychology of Programming*, J.-M. Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore (Eds.). Academic Press, London, Chapter Chapter 1.1, 9–19. https://doi.org/10.1016/B978-0-12-350772-3.50006-9

[117] Luciana Pangrazio. 2016. Reconceptualising critical digital literacy. *Discourse: Studies in the Cultural Politics of Education* 37, 2 (2016), 163–174. https://doi.org/10.1080/01596306.2014.942836

[118] Seymour Papert. 1987. Information Technology and Education: Computer Criticism vs. Technocentric Thinking. *Educational Researcher* 16, 1 (1987), 22–30. https://doi.org/10.3102/0013189X016001022 also: M.I.T. Media Lab Epistemology and Learning Memo No. 1 (November 1990).

[119] Seymour Papert and Idit Harel. 1991. Situating Constructionism. In *Constructionism*, Seymour Papert and Idit Harel (Eds.). Ablex Publishing Corporation, Norwood, NJ, Chapter 1, 1–11. http://www.papert.org/articles/SituatingConstructionism.html

[120] Jiyong Park and Jongho Kim. 2022. A Data-Driven Exploration of the Race between Human Labor and Machines in the 21st Century. *Commun. ACM* 65, 5 (apr 2022), 79–87. https://doi.org/10.1145/3488376

[121] Nancy Pennington and Beatrice Grabowski. 1990. The Tasks of Programming. In *Psychology of Programming*, J.-M. Hoc, T.R.G. Green, R. Samurçay, and D.J. Gilmore (Eds.). Academic Press, London, Chapter 1.3, 45–62. https://doi.org/10.1016/B978-0-12-350772-3.50008-2

[122] Alan J. Perlis and Klaus Samelson. 1958. Preliminary report - International algebraic language. *Commun. ACM* 1, 12 (1958), 8–22. http://www.softwarepreservation.org/projects/ALGOL/report/Algol58_preliminary_report_CACM.pdf

[123] Jacob Perrenet and Eric Kaasenbrood. 2006. Levels of Abstraction in Students' Understanding of the Concept of Algorithm: The Qualitative Perspective. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITICSE '06)*. Association for Computing Machinery, New York, NY, USA, 270–274. https://doi.org/10.1145/1140124.1140196

[124] Mark Priestley. 2011. *A Science of Operations: Machines, Logic and the Invention of Programming.* Springer, London, UK.

[125] Mark Priestley. 2021. Logic, Code, and the History of Programming. *IEEE Annals of the History of Computing* 43, 4 (2021), 92–96. https://doi.org/10.1109/MAHC.2021.3127289

[126] Giuseppe Primiero, Nicola Angius, and Raymond Turner. 2021. The Philosophy of Computer Science. In *Stanford Encyclopedia of Philosophy*. Stanford University, Stanford, CA. https://plato.stanford.edu/entries/computer-science/

[127] William J. Rapaport. 2005. Philosophy of Computer Science: An Introductory Course. *Teaching Philosophy* 4 (2005), 319–341. Issue 28.

[128] Mitchel Resnick and Brian Silverman. 2005. Some Reflections on Designing Construction Kits for Kids. In *Proceedings of the 2005 Conference on Interaction Design and Children* (Boulder, Colorado) *(IDC '05)*. Association for Computing Machinery, New York, NY, USA, 117–122. https://doi.org/10.1145/1109540.1109556

[129] Geoffrey G. Roy. 2006. Designing and Explaining Programs with a Literate Pseudocode. *J. Educ. Resour. Comput.* 6, 1 (mar 2006), 1–es. https://doi.org/10.1145/1217862.1217863

[130] Dana Saito-Stehberger, Leiny Garcia, and Mark Warschauer. 2021. Modifying Curriculum for Novice Computational Thinking Elementary Teachers

and English Language Learners. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*. Association for Computing Machinery, New York, NY, USA, 136–142. https://doi.org/10.1145/3430665.3456355

[131] Shima Salehi, Karen D. Wang, Ruqayya Toorawa, and Carl Wieman. 2020. Can Majoring in Computer Science Improve General Problem-Solving Skills? In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, New York, NY, USA, 156–161. https://doi.org/10.1145/3328778.3366808

[132] Carsten Schulte. 2008. Block Model: An Educational Model of Program Comprehension As a Tool for a Scholarly Approach to Teaching. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. ACM, New York, NY, USA, 149–160. https://doi.org/10.1145/1404520.1404535

[133] Carsten Schulte. 2008. Duality Reconstruction — Teaching Digital Artifacts from a Socio-technical Perspective. In *Proceedings of the 3rd International Conference on Informatics in Secondary Schools - Evolution and Perspectives: Informatics Education - Supporting Computational Thinking* (Torun, Poland) *(ISSEP '08)*. Springer-Verlag, Berlin, Heidelberg, 110–121.

[134] Carsten Schulte. 2012. Uncovering Structure behind Function: The Experiment as Teaching Method in Computer Science Education. In *Proceedings of the 7th Workshop in Primary and Secondary Computing Education* (Hamburg, Germany) *(WiPSCE '12)*. Association for Computing Machinery, New York, NY, USA, 40–47. https://doi.org/10.1145/2481449.2481460

[135] Robert Sedgewick and Kevin Wayne. 2008. *Introduction to Programming in Java: An Interdisciplinary Approach*. Addison-Wesley, Boston, MA.

[136] Sue Sentance, Jane Waite, and Maria Kallia. 2019. Teaching computer programming with PRIMM: a sociocultural perspective. *Computer Science Education* 29, 2-3 (July 2019), 136–176. https://doi.org/10.1080/08993408.2019.1608781 Publisher: Routledge.

[137] Mary Shaw. 1985. The Nature of Computer Science. In *The Carnegie-Mellon Curriculum for Undergraduate Computer Science*. Springer, Cham, Switzerland, 7–12.

[138] IEEE Computer Society. 2012. *IEEE Standard for System and Software Verification and Validation*. Technical Report IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004). IEEE. https://doi.org/10.1109/IEEESTD.2012.6204026

[139] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *Trans. Comput. Educ.* 13, 2 (July 2013), 8:1–8:31. https://doi.org/10.1145/2483710.2483713

[140] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Trans. Comput. Educ.* 13, 4, Article 19 (nov 2013), 40 pages. https://doi.org/10.1145/2534973

[141] Gerald J. Sussman. 2004. The Legacy of Computer Science. In *Computer Science: Reflections on the Field, Reflections from the Field*, Committee on the Fundamentals of Computer Science: Challenges, Computer Science Opportunities, and National Research Council Telecommunications Board (Eds.). The National Academies Press, Washington, DC, 180–183. This can also be found in the coauthored SICP book.

[142] Doron Swade. 2009. Inventing the User: EDSAC in Context. *Comput. J.* 54, 1 (12 2009), 143–147. https://doi.org/10.1093/comjnl/bxp116

[143] Christian Swertz. 2022. The Message of the Method: Design Principles for Sustainable Ideological Communication in the Digital Classroom – Keynote of the 20th DELFI-Tagung. Delfi 2022 – Fachtagung Bildungstechnologien der Gesellschaft für Informatik e. V.. Henning, Peter A. and Striewe, Michael and Wölfl, Matthis (Hrsg.).

[144] Laurence Tamatea. 2019. Compulsory coding in education: liberal-humanism, Baudrillard and the 'problem' of abstraction. *Research and Practice in Technology Enhanced Learning* 14, 1 (2019), 1–29. https://doi.org/10.1186/s41039-019-0106-3

[145] Kumiko Tanaka-Ishii. 2010. *Semiotics of Programming* (1st ed.). Cambridge University Press, USA.

[146] Matti Tedre. 2014. *The Science of Computing: Shaping a Discipline*. CRC Press, Boca Raton, FL.

[147] Matti Tedre. 2018. The Nature of Computing as a Discipline. In *Computer Science Education – Perspectives on Teaching and Learning in School*, Sue Sentance, Erik Barendsen, and Carsten Schulte (Eds.). Bloomsbury Publishing, London, UK, Chapter 1, 5–18.

[148] Josh Tenenberg. 2001. On the Meaning of Computer Programs. In *Cognitive Technology: Instruments of Mind*, Meurig Beynon, Chrystopher L. Nehaniv, and Kerstin Dautenhahn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 165–174.

[149] Josh Tenenberg and Yifat Ben-David Kolikant. 2014. Computer Programs, Dialogicality, and Intentionality. In *Proceedings of the Tenth Annual Conference on International Computing Education Research* (Glasgow, Scotland, United Kingdom) *(ICER '14)*. Association for Computing Machinery, New York, NY, USA, 99–106. https://doi.org/10.1145/2632320.2632351

[150] Karin Tengler, Oliver Kastner-Hauler, and Barbara Sabitzer. 2021. Enhancing Computational Thinking Skills using Robots and Digital Storytelling. In *Proceedings of the 13th International Conference on Computer Supported Education (CSEDU 2021, Vol. 1)*. SCITEPRESS, Nijmegen, Netherlands, 157–164. https://doi.org/10.5220/0010477001570164

[151] Rob Thompson and Steve Tanimoto. 2016. Children's Storytelling and Coding: Literature Review and Future Potential. In *Proceedings of the 27th Annual Workshop of the Psychology of Programming Interest Group – PPIG 2016*. Psychology of Programming Interest Group, University of Cambridge, UK, 9 pages.

[152] Michael Thuné and Anna Eckerdal. 2009. Variation theory applied to students' conceptions of computer programming. *European Journal of Engineering Education* 34, 4 (2009), 339–347. https://doi.org/10.1080/03043790902989374

[153] Preston Tunnell Wilson, Kathi Fisler, and Shriram Krishnamurthi. 2018. Evaluating the Tracing of Recursion in the Substitution Notional Machine. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. ACM, New York, NY, USA, 1023–1028. https://doi.org/10.1145/3159450.3159479

[154] Sherry Turkle and Seymour Papert. 1990. Epistemological pluralism: Styles and voices within the computer culture. *Signs: Journal of women in culture and society* 16, 1 (1990), 128–157.

[155] Raymond Turner. 2018. *Computational Artifacts: Towards a Philosophy of Computer Science*. Springer, Cham, Switzerland.

[156] Ian Utting, Stephen Cooper, Michael Kölling, John Maloney, and Mitchel Resnick. 2010. Alice, Greenfoot, and Scratch – A Discussion. *ACM Trans. Comput. Educ.* 10, 4, Article 17 (nov 2010), 11 pages. https://doi.org/10.1145/1868358.1868364

[157] Henry M. Walker. 2011. Resolved: ban 'programming' from introductory computing courses. *ACM Inroads* 2, 4 (2011), 16–17.

[158] Eugene Wallingford. 1998. Elementary patterns and their role in instruction. In *OOPSLA'98*. ACM, Vancouver, British Columbia, Canada.

[159] Peter Wegner. 1976. Research Paradigms in Computer Science. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, California, USA) *(ICSE '76)*. IEEE Computer Society Press, Washington, DC, USA, 322–330.

[160] Michael Weigend. 2006. From Intuition to Programme. In *Informatics Education – The Bridge between Using and Understanding Computers*, Roland T. Mittermeir (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 117–126.

[161] Ben Williamson. 2016. Political computational thinking: policy networks, digital governance and 'learning to code'. *Critical Policy Studies* 10, 1 (2016), 39–58. https://doi.org/10.1080/19460171.2015.1052003

[162] Jeannette M. Wing. 2010. Research Notebook: Computational Thinking – What and Why? *The Link Magazine* 6 (2010), 20–23. Issue Spring. https://www.cs.cmu.edu/link/research-notebook-computational-thinking-what-and-why

[163] Aman Yadav, Marie Heath, and Anne Drew Hu. 2022. Toward Justice in Computer Science through Community, Criticality, and Citizenship. *Commun. ACM* 65, 5 (apr 2022), 42–44. https://doi.org/10.1145/3527203

[164] Heinz Zemanek. 1966. Semiotics and Programming Languages. *Commun. ACM* 9, 3 (mar 1966), 139–143. https://doi.org/10.1145/365230.365249

[165] Yifan Zhang, Amanda Mohammad Mirzaei, Lori Pollock, Chrystalla Mouza, and Kevin Guidry. 2021. Exploring Computational Thinking Across Disciplines Through Student-Generated Artifact Analysis. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 1315. https://doi.org/10.1145/3408877.3439594 event-place: Virtual Event, USA.

[166] Cheng Zhou, Sandeep Kaur Kuttal, and Iftekhar Ahmed. 2018. What Makes a Good Developer? An Empirical Study of Developers' Technical and Social Competencies. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Press, Lincoln, NE, USA, 319–321. https://doi.org/10.1109/VLHCC.2018.8506577

A THE "NATURE OF PROGRAMS" FRAMEWORK

The Nature of Programs

A framework for educators

ITiCSE 2022 - Working Group 5

What is a computer program? This may seem to be a trivial question, yet programs are strange creatures with a multifaceted nature that eludes simple definitions.
The purpose of this document is to provide educators and policy makers with a broad, integrated view of the Nature of Programs, which can permeate the design and implementation of classroom activities aimed at introducing students to the basic principles of computer science and programming.
In the first part of the document, we describe the nature of programs through 6 different facets, each showing a different way of understanding what a program is. In the second part we discuss what is involved in the process of creating and developing programs, beyond simply "writing code," and relate the concept of *program* to other central concepts in programming and computer science.

Table of contents

Introduction

Computer programs permeate our lives more than a lot of people might recognise. Every app and digital device that we use involves running computer programs that have been developed for a purpose:

- the alarm that wakes us;
- the kitchen appliance that we use to prepare breakfast;
- sharing events in our lives with our social networks;
- our decision about what to wear based on a weather forecast;
- the transport (public or private) we take that is controlled by computers;
- the tasks we accomplish during the day, which could include using email, word processing, computer-controlled tools, digitally delivered plans, and mobile communication;
- a point of sale terminal in a shop;
- a feature on a mobile device that makes it work as a phone or a radio;
- and much more!

In other words, programs have steadily become ubiquitous and, in many cases, "seamless": they are there but you cannot see them.

So why are computer programs so widely used, and what makes them so versatile and powerful? Is there value in students being able to understand what is behind these apps that regulate and control our everyday lives? And what does programming involve?

Answering these questions is not straightforward, as the multifaceted nature of the programs makes it difficult to summarise their key features in a few words. Instead, there are so many different aspects of programming that it is valuable to explore them in depth, as we will do in these pages. We are particularly interested in doing this so that educators can develop a broad view of what programs are and how they are created.

In Part I we describe what a computer program is using six different *facets* that show complementary ways of understanding what a program is. In Part II we investigate, without committing to any specific software development methodology, what the creation of programs entails, in order to broaden the understanding beyond simply "writing code".

The term "program" is used here in a very broad sense, ranging from simple programs that are studied in introductory CS classes to complex software systems or services such as an operating system or a social network. In fact we believe that all these share the same nature, as described and argued in the six facets description, despite their differences in their size, complexity, and user involvement (while acknowledging that those differences have a very important role, especially considering the development process and the impact on their users).

We restrict ourselves to programs that one could call "traditional". That is, our focus is on programs that are an implementation of human designed algorithms, which consequently allows us to explain how they achieve their intended goal. Such a decision leaves out of our scope programs whose implemented logic is not human made but machine produced
(e.g. by evolutionary programming or machine learning techniques).

Part I: Facets of programs

Programs are many things to many people – to some they are tools important for their day-to-day work, or even have created jobs that didn't exist a few years ago (such as YouTubers, Influencers, or Data Scientists); to others they are artefacts worthy of study in their own right; and to others they are a threat to the environment and society. In education, programming can be an exercise in problem solving that may open up whole new possibilities, or it might simply be an interesting intellectual exercise.

The motivations and methods for teaching students to program also vary depending on which of the above views are emphasised. Our goal is to thoroughly explore what a program entails so that those involved in education (both teachers and students) can develop a richer view of what this core component of computer science is about.

In order to reflect such a broad view, we have identified six facets, which together give us a diversified understanding of computer programs, as follows:

- Programs are used as tools
- Programs are human-made technology
- Programs are physical objects
- Programs are abstract entities
- Programs are executable entities
- Programs are notational artefacts

It is because programs involve all of these different facets that they can be differentiated from other technologies and ideas.

Figure 1 shows visually the six different facets of programs that we will discuss.

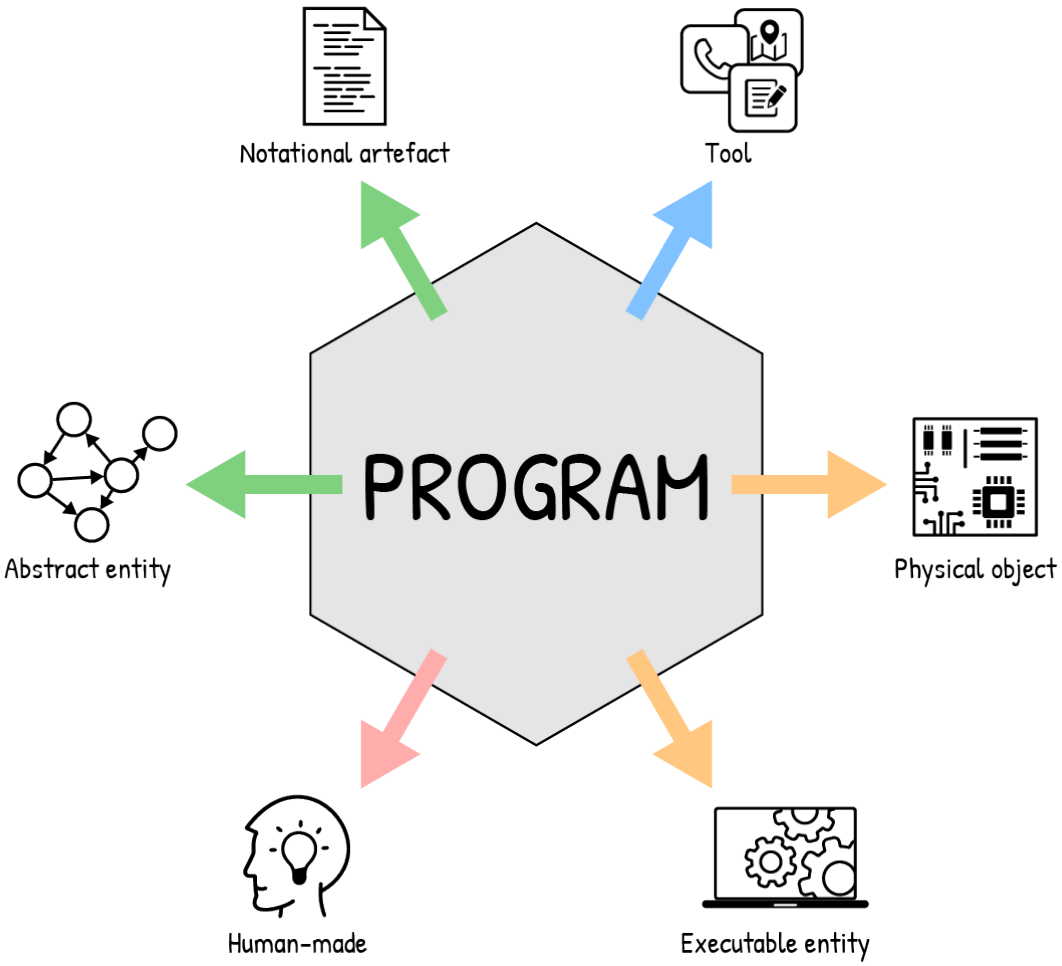


Fig. 1: Diagram showing the six programs facets and their interrelations (the colours of the arrows reflect the concept maps in Part 2).

The facets are complementary, and even sometimes apparently contradictory. In addition to exploring each of the six facets, we will discuss three dichotomies between pair of facets that are at the opposite sides in the diagram: notational vs executable, abstract vs physical, and human made vs being a tool that humans use.

The remainder of this section explores the facets in some detail. Each facet has a description, a discussion of its impact, some examples of how the facet might be observed in action, and for each opposed pair, a brief exploration of the dichotomy it entails. To save time, we encourage you to read the description, and then skim the extra sections for useful supporting ideas.

Facet 1: Tool

For many people using digital devices, a program is a tool to support them in their work and leisure. People depend on tools for their daily activities. Programs (such as photo editing software, an alarm clock, spreadsheet, video game, video editor or a weather prediction system) are used to enable people to work more effectively, to be more creative, and to communicate with a wider audience than they could without the tool.

A program can turn the same physical device (such as a mobile phone) into a messaging system, a notepad, a weather prediction system, a game, or even a camera if proper hardware components are available (such as optical lenses and light sensors). This is made possible by the fact that the devices are programmable, and this flexibility has led to the ubiquity of programs, since new software can be created to add new functions to existing hardware. Thus, it is not unusual that mobile phone users carry dozens of programs with them, making the phone role just one of the many functionalities of the device.

Often these tools are taken for granted and are so ubiquitous that they blend into the background. Sometimes they only become visible when something goes wrong. Moreover, tools are not neutral, but embed values and force people to act according to predefined views and expectations.

Impact of programs as tools

Since programs are also *tools*, their impact on so-called users and the way the user is being modelled in a program need to be considered when educating students about computer programs. The computing industry has not only created tools for a diverse range of uses, from military applications to regulating our social lives, but it has also developed implicit interpretations of how a tool might be used, so these programs also reshape the domain of use, resulting in new problems and challenges. For example, social media started as a way to access news about people and events, but the particular ways in which social media provides us with that information means that we are receiving targeted news that has been chosen by an algorithm based on data it has gathered about the user. This makes the news relevant to the user and increases the impact as it is targeted specifically to the user, but it can also have the unintended impact of amplifying misleading information. Moreover, while programs have an intended use while being created, in the hands of users they can develop into a tool in their own right with potential uses beyond what the designers had anticipated.

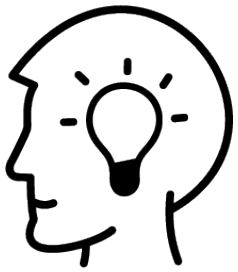
Examples of programs as tools

- Transport such as automating the operation of a car, assisting navigation, organising international travel.
- Supporting people to communicate, including social media, video conferencing, email, mobile messaging.
- Automating shopping, which includes speeding up the checkout process, enabling self checkout, online shopping, and other ways of giving the customer empowerment over the process.

NoP Framework: page vi

- Recreation, including games, entertainment, creation and viewing of films, and creating and listening to music.
- Managing our environment, including energy access and use, heating, lighting, and security systems.
- Apps created for niche purposes, such as rain radars, or knowing parking restrictions in local streets.

Facet 2: Human-made



People make programs to satisfy human needs, including induced ones, or the need to express oneself. One could consider the phenomena of nature itself as systems that process information (think of DNA transcription or plants that use chemicals to transmit information). But, unlike these systems, computer programs are built by humans intentionally to exploit information processing devices.

Programs are made as tools with a purpose not only for the user (such as providing them with a good text processor) but also the maker (to earn a living or acquire information). Commonly a program is made by teams of people, and the systems developed may consist of many interacting programs, so to be good at making this technology, social skills come into play very quickly (such as teamwork, communication, collaboration, creativity, problem solving, and critical thinking). A developer needs to be able to discuss options, argue for different solutions to a problem, and work with others to create a suitable solution. Many steps of this process use software tools themselves to make programming more efficient, but the key driver is a person or people with an idea.

Because programs are human-made they are not value-free. The values, ideas and ways of thinking of developers and businesses that are creating programs are reflected either explicitly or implicitly in a program and the ways it can be used. Moreover, since programs are made by humans, and humans are not infallible, one needs to keep in mind that programs too may contain errors.

Impact of programs as human-made

Programs are human made, which means that they embed the ideologies, intentions and ways of thinking of the developers and the organisations that make programs available. It is important for children to be aware of this facet. For instance, it is important to realise that when they are using an App on their phone for "free", the group of people who created that program might be gaining income in another way, perhaps by collecting the children's data and selling it to some other business, or re-using that data to feed to their machine learning programs. On the other hand, "free software" (using "free" in the sense of "freedom of speech") can help level the playing field, as it is an efficient way of sharing ideas and avoiding power being held by particular organisations.

Moreover, the very fact that in order to make programs you need humans who are well-versed in making programs (designing, coding, maintaining, testing and so on) has haunted the field of programming since the 1950s: the shortage of skilled programmers and the consequent high costs involved in hiring programmers has limited the rate at which software can be produced, and has made a programming career very attractive as a skill in limited supply. It has also been a driving factor to develop programming tools that automate the making of programs, ranging from better development environments to the idea of low-code or no-code systems that support rapid development (but are themselves tools that programmers must develop!)

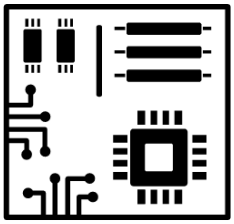
Examples of programs as human-made

- Thousands of new "apps" are released for mobile phones every month. To justify the work needed to create them, they need to reflect an innovation where someone has identified an audience that would benefit from their app.
- The Strategic Defense Initiative (Star Wars) that was launched in the U.S. by Ronald Reagan in the 1980s resulted in a public scandal when David Parnas, a well-known software engineer, resigned from the project because the requirements of the software were unrealistic. This was due to, amongst other things, the fact that it would be impossible to be sure that it would not contain any errors. In this kind of setting, software errors can result in catastrophe.
- There have been some very costly consequences of human error in the production of software including the failed maiden flight of Ariane 501, the millennium bug, and the mishandling of baggage by the new software system at Heathrow's terminal 5.
- The world view and limited experience of programmers behind a particular program might lead to a bias being embedded in the final product, such as assuming good eyesight in the users, or an understanding of how to use a particular kind of interface.
- The Scratch programming environment is an example of how software systems embed a certain idea or conception of "need". Scratch is today one of the main environments in which children learn to "code," but given its particular vision on computer science education it also gives a very particular experience of "programming" that might be misleading to young children.

Dichotomy of programs as tools vs human-made

A large part of the call for students to learn programming is to enable them to be "creators" rather than just "users" of digital technologies. Computer programs are valuable tools, and educating students on using these tools (such as search engines, spreadsheets, and video editors) is important, since these enable them to be productive and creative. But if they are limited to only being the users of this tool, and don't appreciate that they are made by people who are skilled at programming, they lose the opportunity to see programming as a career pathway for themselves, or at least realising where programs come from and what is involved in creating them. This has led to teaching programmes based on a "use-modify-create" approach, where students explore existing programs, but grow into being able to create their own.

Facet 3: Physical object



Even though software is an abstract concept, programs are stored on a physical medium and they can be executed, read, copied, changed or even subject to corruption. Moreover, their execution takes time and uses energy, although unlike most other familiar physical objects, consumed time, energy and space are usually on an extremely small scale.

The physical facet might seem secondary, but as a novel does not exist without a physical medium carrying its message, a program is not just in an imaginary cloud – in fact, even if it is running on a virtual machine in "the cloud", it is using energy and space at a particular location.

It is possible that programs can also be executed by humans (for instance, painstakingly tracing a program's operation on a piece of paper) and they can also be stored in a physical medium that is not a machine per se (for instance, flow diagrams from the 1950s; or instructions on paper). However, usually, when we speak about programs we expect them to be executed on a machine, which can execute millions of instructions reliably in a very short amount of time.

Impact of programs as physical objects

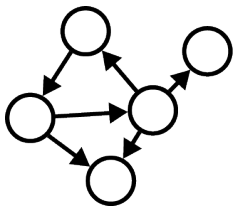
Programs that are run on a computing device consume space, energy and time, and the construction of the device uses resources such as labour and precious metals. These observations in themselves may raise concerns. It is important for students to realise that to run a program, there is an impact in the "real world," which also includes social and psychological implications.

In addition to the issues around using personal devices to run a program, it is also important that students understand that something "in the cloud" or "on the web" always has a physical location to which certain laws apply that might be very different from the laws one would assume apply. Having a cloud service based in Europe for instance gives much better protection of data than having it located in the US, thanks to the EU GDPR. The observation that programs consume energy and resources while they are running is of greater concern today than it was 20 years ago because of the increasing impact on the environment.

Examples of programs as physical objects

- Apps are commonly stored on flash memory in mobile devices, and software is typically stored on hard drives in desktop computers; in both cases, these are programs, but the name used to refer to them has diverged depending on the context in which they are used.
- Programs are moved to primary memory (and other types of memory such as caches) to allow them to be executed.
- It takes time to download the digital data needed to install or update software.
- Programs use energy in order to run, and the importance of cooling for most computers highlights the environmental impact they may be having.
- Digital devices are built using scarce resources, and building devices to run our programs has sustainability considerations.
- Programs sense and affect the physical environment through input and output, which could be as minor as placing an image on a screen, or as substantial as controlling water flowing through a dam.
- Crypto-currencies such as bitcoin are known to have a huge environmental impact because of the energy-intensive way by which transactions need to be verified, as well as the use of energy for "mining" currency. The impact is so enormous that charities like Greenpeace and WWF have abandoned using crypto-currency.

Facet 4: Abstract entity



Abstraction is fundamental to programs and programming. In the same way that a story, joke, idea or concept is abstract, a program is something that refers to and manipulates abstract notions and entities. The data that programs manipulate is just digits, which are an abstract concept even if they are used to represent something physical in the real world, whether it is the amount of money that someone has in their bank, the colour of a pixel, a recording of some music, or even something that itself is abstract, like a story. Moreover, the particular ways in which numbers are represented in the machine are always an abstraction of the physical and continuous machinery underlying them. The program eventually determines a series of actions that the system executes, and as a result the program determines physical effects in the system itself (such as the device memory). However, what is relevant is not these physical effects as such, but the fact that they can be interpreted according to the abstractions the program manipulates, such as a number, a portion of a picture, or a sound.

Programs implement algorithms, and an algorithm is not physical – it's an idea of how to do something. An algorithm captures some abstract view of a class of programs, and like other abstract concepts it is a construction of someone's mind (and not necessarily the same construction for all minds). The only way to describe an algorithm to others, however, is to draft a more or less formal/detailed program in some (written or oral) language, which is strongly connected to the "notational entity" facet described later. In this sense, we can see a program as a particular incarnation of a decidedly abstract idea.

Impact of programs as abstract entities

Because abstraction is such an important part of programming, students will need to learn to seek out the core elements of a problem, and decide which details should be modelled in order to solve it. This inevitably means there are many ways to achieve the same general outcome; for example, to represent a date such as 11 July 2022, it might be abstracted to three numbers (11, 7, 2022) or the number of days since 1 Jan 1900 (44,753), or any of several other formats.

The abstraction surrounding programming also means that an algorithm doesn't have exactly one program that it corresponds to; in fact, multiple programmers implementing the same algorithm are unlikely to end up with identical programs. More generally, there is no single "correct" program that achieves a particular outcome, whereas students may be used to the idea that normally in problem solving there is just one correct answer. In fact, there is an infinite number of programs that achieve a particular outcome (there is also an infinite number that are incorrect!).

Consequently, if a developer decides to represent a given object in the world in a particular manner, then someone studying the software should be aware that that representation is not absolute but relative to the aims and worldviews of the developer, team of developers or the client. This partly explains issues related to bias in computer programs. In general, since there is no definite rule to

decide on the best abstractions, one might also end up with bad abstractions resulting in malfunctioning or complex programs.

Examples of programs as abstract entities

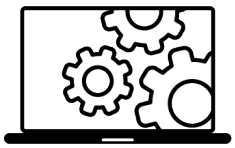
- Simple programming languages for beginners sometimes have instruction sets like "Forward", "Right" and "Left" to program a sprite or a robot to move around. These instructions may abstract away from the fact that the physical movements are never perfectly oriented; even the direction of a robot is subjected to some (maybe imperceptible) errors. However these abstractions are useful to approximate sufficiently well the description of the movements to be carried out.
- When planning how to implement a program, intermediate approaches (such as informal descriptions, flow charts or pseudo-code) are useful to help the programmer work at an appropriate level of abstraction and not become overwhelmed with the detail of the implementation.
- When we name variables in a programming environment they often suggest a view of a program object as a metaphor of something from the real world. We could for instance define a class of objects known as "person", and give these a number of properties (for example "name" and "identity number"). This class of objects abstracts what the programmer needs to know about a person for their context, and a "real" person is far more than a name and number!
- The way numbers are represented in a computer is a basic concept in numerical analysis. Since "real numbers" in a programming environment can only be finite approximations of "actual" real numbers, programmers working in a modelling context need to be very careful about potential issues like rounding errors.

Dichotomy of programs as physical objects vs abstract entities

It may seem odd that we have claimed that a program is both abstract and physical! Although apparently antithetical to each other, concrete and abstract traits usually coexist when we think about programs. On the one hand, we usually make sense of a program's behaviour by envisaging a world of abstractions that somehow come alive in our mind, thus actually dealing with an abstract entity. On the other hand, it is also a concrete physical object as soon as we code, manipulate and run it by means of a computing device. Because of this sort of ambivalence, computer science has even been defined as "the discipline of concrete abstractions"[1].

[1] Max Hailperin, Barbara Kaiser, and Karl Knight. 1999. *Concrete Abstractions*. Brooks/Cole, Pacific Grove, CA

Facet 5: Executable entity



A program can be executed, that is, it can be run on a physical device indefinitely, at different times by different users. A program can be characterised as something that is automatically executed by an information-processing agent, where the agent unquestioningly follows instructions with no understanding of the purpose of the set of instructions it has been given. Any decisions that are made by a program are already programmed into it by the programmer. Once a user downloads and starts using a program, the programmer no longer has control; every response to the user's interaction and input data is determined by what is already in the program.

The automation is based on a finite and very precise set of instructions that a digital device is capable of executing. At the lowest level the program is executed using machine instructions for the particular processor in use, although usually these instructions are themselves generated from tools that enable programmers to work in a high-level language that itself has a well-defined instruction set.

Impact of programs as executable entities

Because the execution of a program is automatic, in typical commercial situations the programmer(s) release the program to users, at which point the programmer no longer has control over what will happen, and in many cases, may not even know when the program is being used. This emphasises the importance of testing -- before software is released, it needs to be tested to ensure that it will function well for the large variety of ways in which it may be used. Students can lose sight of this because as beginners they are often the programmer, tester *and* user for their own programs, which unfortunately means that as a user they already understand the program well and may not see it from a more typical user's point of view. Having someone other than the programmer test a program (including its user interface) can be valuable for beginners to give them insight into how it will be seen by others, and means that they can program it so that an automated version is usable and useful.

The fact that a program that is executed on a computing machine needs to be automatic while being capable of handling the largely unpredictable behaviour of user interactions or other interactions with the external environment, results in the problem of developing programs that control future behaviours that cannot be mapped out in full detail. It is here then that the problem of finding the right representations of the world inside the program is critical for the program to function in a satisfying manner.

In practice, this is particularly important if the software is controlling critical processes such as aircrafts, spaceships, financial systems and nuclear power plants, where the consequences of not anticipating what will happen in the automation could be catastrophic.

Conversely, this also means that one programmer can write a program that can be used by millions of people, and for long periods of time, which creates an economic model that is quite different to conventional manufacture and distribution.

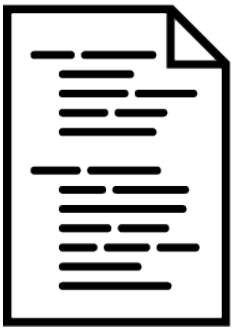
Examples of programs as executable entities

- People can use a program without understanding how it works – the programmer has created a tool in the form of a program that the user can wield if they have a device that will execute the program for them, but the user can treat the program as a black box. In fact,

much commercial software intentionally makes the program itself opaque to the user, since the code is a trade secret.

- The computer running the program is unaware of the program's purpose; it follows instructions that come from a well-defined instruction set by strictly doing what each instruction requires.
- If an interaction with the program occurs that was not anticipated by the programmer, it can result in malfunctioning programs. This is of particular importance in safety-critical systems like self-driving cars.
- Older examples of automatic (programmed) machines are Jacquard's Loom (digital versions of which are still in use) and music boxes (again, still in use to some extent).

Facet 6: Notational artefact



A program is a notational artefact, in the same way that a manuscript, book, or music score is: it relies on a notation with a particular syntax, according to some formal rules, linguistic notations and conventions. Just like the writing of a book by humans usually involves some creativity and tacit knowledge, the same holds true of programs made by humans. Programs can exist in their own right as an object of interest, but they are mainly used to communicate (from programmers to machines, or among programmers) about the processes to be carried out to achieve some goal.

When intended to be communicated to machines, programs rely on formal languages that can be automatically parsed and executed. However since they are also intended for human programmers, they typically include many non-formal linguistic elements, such as meaningful variable names, clear and precise comments, interface translations, and supporting documentation. All of these elements work together to make the program useful and efficient for people to work on.

Another feature of programm notations intended for the machine is that they can always be converted to another notation using an automated "translator" (for instance a compiler or an interpreter). The idea that one can always convert a notation into another one has been a foundational insight of programming language design since the 1950s. Indeed, it is because of this very possibility that we can abstract away from the computer hardware to present programs in a notation that is also understandable and usable by humans.

Impact of programs as notational artefacts

Program scripts are written for people. The program itself is usually intended for an end-user, who may have received a version of the script that has been converted to an executable app, and have no interest in the notation it represents. But programs are also written for the next programmer who may need to modify or debug it. The "next programmer" may be the original author who needs reminding how the program was designed, since programs are often of sufficient complexity that it is hard to fully understand them. Thus, writing good programs requires written communication skills that make the program easy to understand for a human. Even choosing a good name for a function or a variable is a challenge; if a variable name is too short then its meaning is unclear, and if it is too long it leads to hard-to-read code; choosing short, meaningful and unambiguous wording for variable names and comments is an important skill.

Programs express ideas that enable us to articulate our thoughts and share them with others. One author went as far as to say that "a computer language is not just a way of getting a computer to perform operations but rather it is a novel formal medium for expressing ideas about methodology. Thus programs must be written for people to read, and only incidentally for machines to execute."[2]

[2] Gerald Jay Sussman. 2004. *The Legacy of Computer Science*. In Computer

Examples of programs as notational artefacts

- It is important to wield programming notation effectively. The structure of a program can affect its readability - badly structured code (including "spaghetti" code) is hard to follow, and refactoring programs is important to keep programs understandable
- Writing a concise comment that explains how a program works, or choosing a variable name that precisely explains its purpose, is a **literacy skill** that is key to successful programming.
- The notation used for programs conforms to formal languages that are well defined, and avoid the ambiguity of natural languages.
- There are awards for authors of deliberately obfuscated programs! They are a kind of art form, but are not useful for getting things done in conventional programming, and generally highlight the impediment that a badly written program gives to those trying to understand it.

Dichotomy of programs as executable entities vs notational artefacts

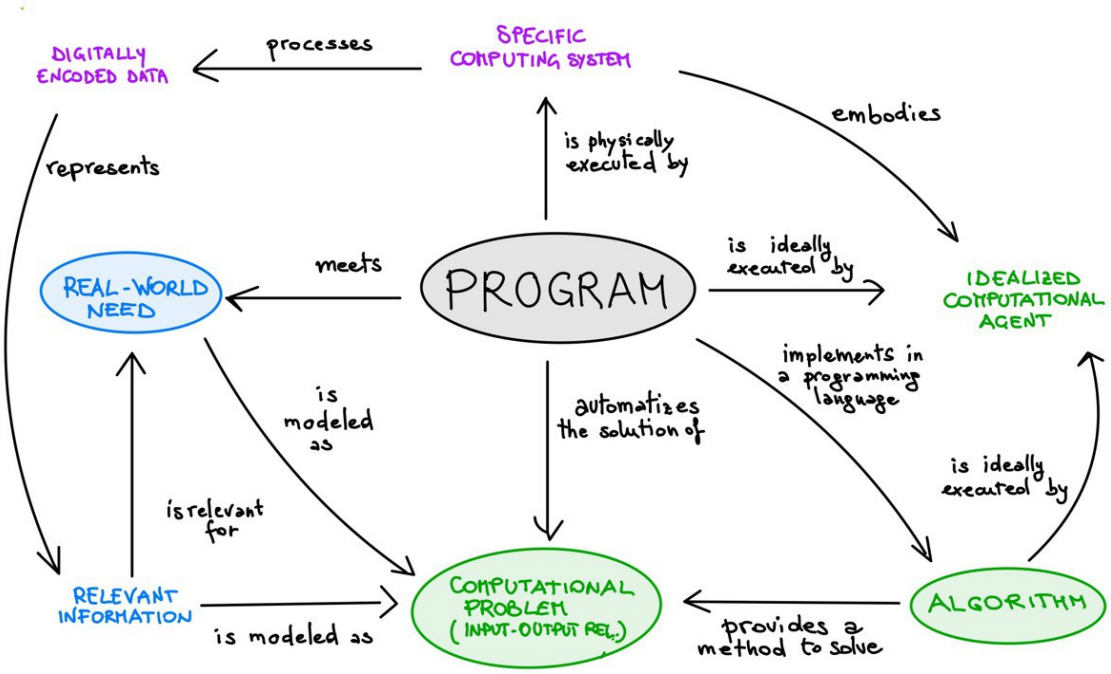
The idea of programming came about in order to automate tasks, and so inevitably it is important that a program can be executed. However, we now have a multitude of programming languages that offer a variety of notations for creating programs, and the notation has become an important aspect of study in its own right. Because a program runs automatically, there is no need for the end user to understand the notation. Conversely, the notation can be used simply as a way to express ideas, and elements of it (such as variable names and comments) can be more important for communicating the ideas to other programmers than affecting the execution of the program. In extreme cases, a program may be written primarily as a work of art (such as "literate programming" or "live coding") or even for humour (such as obfuscated code contests).

Science: Reflections on the Field, Reflections from the Field, Committee on the Fundamentals of Computer Science: Challenges, Computer Science Opportunities, and National Research Council Telecommunications Board (Eds.). The National Academies Press, 180–183. This can also be found in the co-authored book: Harold Abelson, Gerald J. Sussman, and Julie Sussman. 1984. *Structure and Interpretation of Computer Programs*. The MIT Press, Cambridge, MA.

Part II: How programs are created

In this part we use the following concept map to explore how programs come about, and their relationship to algorithms, computing systems, and the real world.

A concept map



Programs are written to *describe the automation of the solutions* to **computational problems**. Computational problems are the information-processing counterpart of real-world needs or problems, and they *model* real-world needs by *representing* the **relevant information** with suitable data.

A program can be run automatically, i.e., *physically executed* by a **specific computing system** (e.g., a computer, or a mobile phone). Such systems normally *process* only digital (i.e. symbolic) encodings of data, hence the data must be **digitally encoded**, i.e. represented as sequences of symbols.

An **algorithm** *provides a method* (i.e., a precise procedure that can be carried out in a prescriptive way) to solve a computational problem. The algorithm is designed to be *executed* by an **idealised computational agent**, capable of executing a predefined set of actions. An actual computing system instantiates such an idealised computational agent.

In order to be automatically executed by an actual computing system, an algorithm must be *implemented*, i.e., written *in a programming language* according to strict rules and complying to the specific computing system's constraints, thus resulting in a program.

Example: grocery checkout

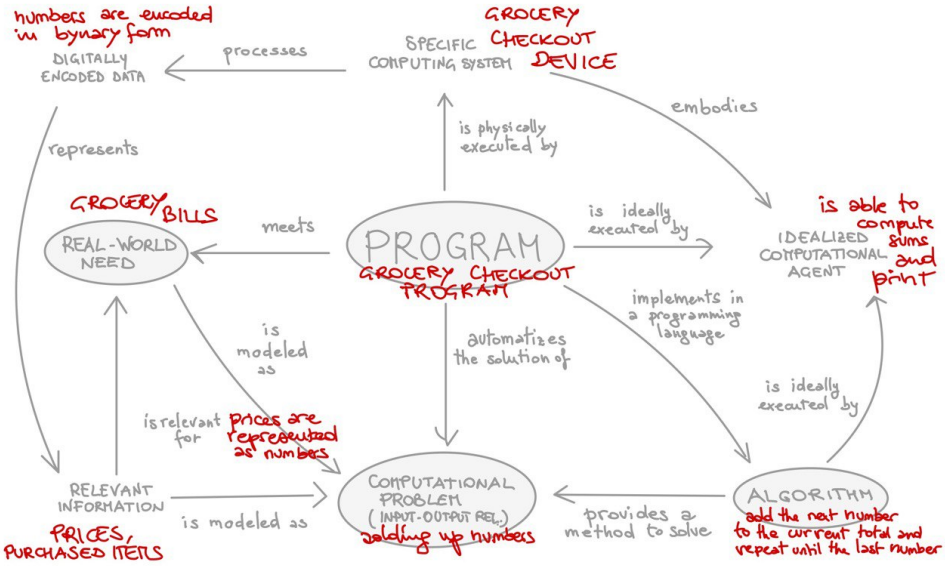
This example is used to illustrate how the diagram above would apply to a specific situation (an annotated version is given below).

A grocery checkout program is meant to support shopkeepers who need to compute grocery bills for their clients. The prices of the purchased items can be modelled as numbers. Thus, computing the grocery bill can be described as the general computational problem of "adding up a series of numbers".

The grocery checkout program is run by a checkout device, which can read numbers by scanning barcodes and print the result on paper receipts. The device actually manipulates numbers encoded in binary form.

The typical algorithm to add up a series of numbers is the following: 1) use a variable to keep track of the current total and set it initially at zero; 2) until you haven't finished the numbers repeat the following instruction: add the next number to the variable that keeps track of the current total; 3) at the end print the variable that keeps track of the current total. When designing this algorithm, we assume that its executor can ideally input any number, sum up two any numbers, and print any number. The grocery checkout device executes such instructions physically. This implies some constraints, for instance, that the device can process only a limited range of numbers.

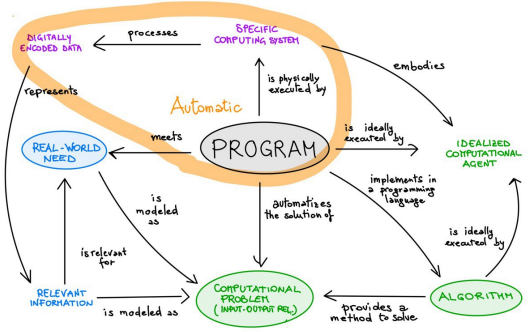
The algorithm is probably implemented in a programming language specialised for checkout devices. However, both the computational problem and the algorithm can suit other similar real-world needs, such as computing the total population of a country based on knowing the population of each of its regions. Hence, if implemented in a general-purpose programming language, the program could be used for different contexts as well.



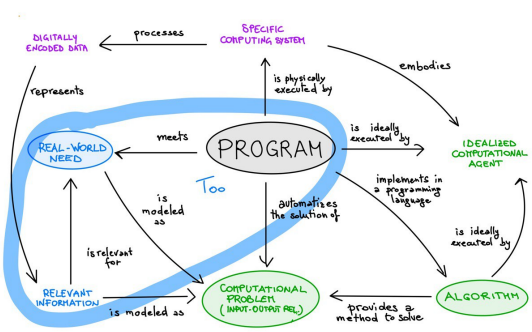
Connection between concept map and program facets

The following diagrams show the aspects that are most relevant to the facets described in part 1 of this document. The last one is described in more detail below.

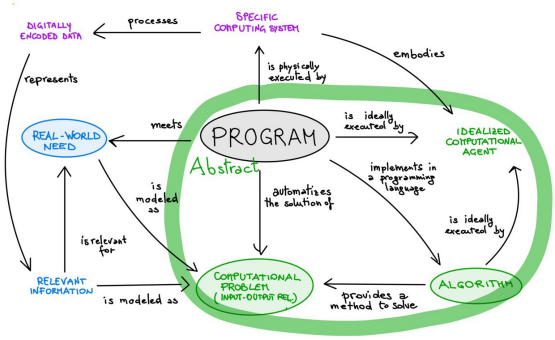
Automatic / Physical



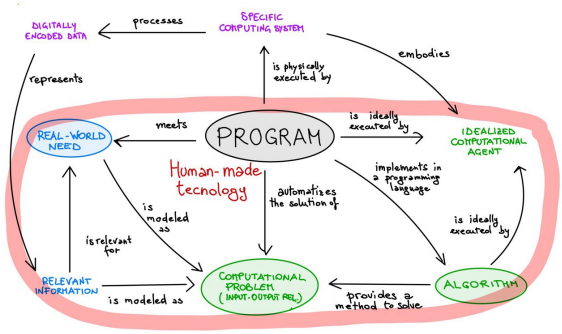
Tool



Abstract / Notational

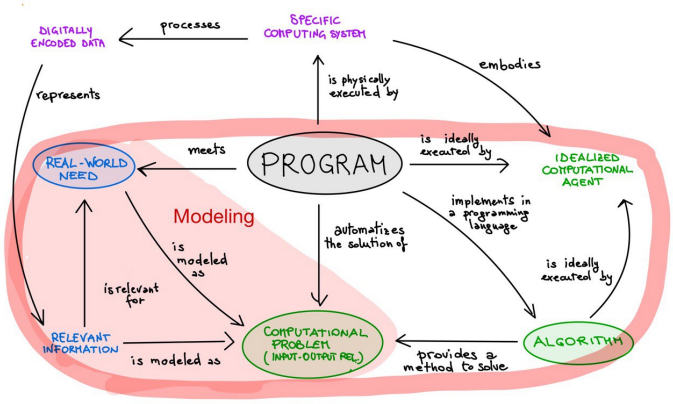


Human-made technology



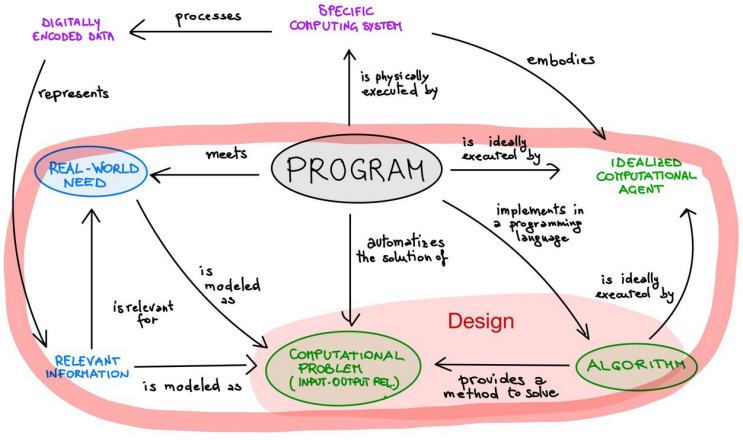
Programs and programming

Programs are a human-made technology. *Programming* is the process of creating and developing programs. There is not one single way to develop programs; instead, different practices are used when programming, in accordance with the type of task the program is automating. However, in the making of any program there are some fundamental processes that typically occur, and that are carried out by humans (usually with the support of some other programs). While conceptually it is worth identifying these processes separately, in practice the distinction among them is much more blurred; thus, the linear order in which we present them should not suggest they must be carried out sequentially; even reiterations or u-turns may be necessary during the overall programming process.



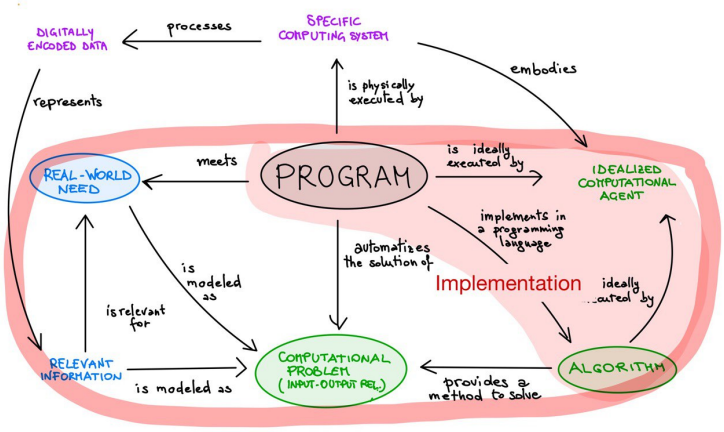
Modeling - Real-world needs and problems are modelled as computational problems. Real-life problems occur with specific instances, but computational problems generally cover a whole set of specific instances. Modelling a real-world problem means understanding the main entities involved and their relationships, and deciding how to express the information relevant to the problem using appropriate data. The input data represents known information, and the output data, once interpreted within the proper context, provides the answer to the problem. It is common to distinguish between data and information, where data are just raw patterns of signs, able to provide information only when interpreted by humans in the context of the problem being solved. A computational problem is specified by defining the relation between the input data and the output data, that is, how the output data depends on the input data. Usually different input data yields different output data. Moreover, the input and output data are not necessarily available at the beginning of execution or produced at the end of the execution, but they can even be part of a continuous interaction among the user, specific sensory units and the computing system.

Design - An algorithm for a computational problem is a description of how to automatically get the output data that is associated – in that computational problem – with any given input data (computer scientists say that the algorithm for a computational problem is a *solution* for that computational problem). A relevant difficulty in designing algorithms is the great number of different cases that can occur during its execution

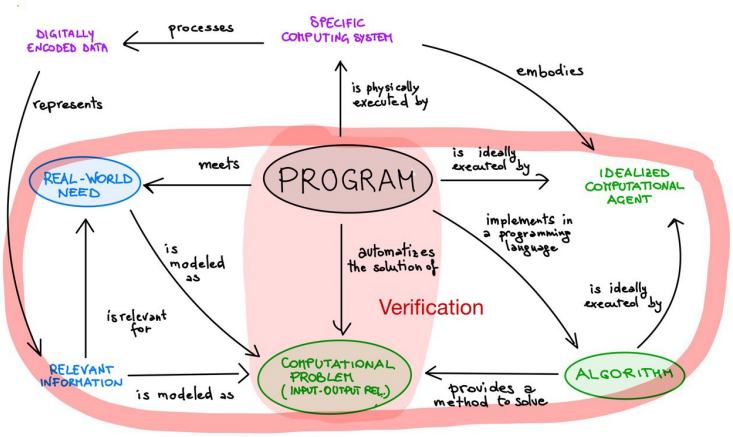


and that have to be considered in advance. Different algorithms may be designed to solve the same computational problem. Various problem-solving techniques can be used to devise and design an algorithm.

Implementation - Algorithms are implemented as programs, that is, they are written *in a programming language* according to strict rules and complying to the computing system's constraints. This activity is sometimes called *coding*. The term *programming* is also often used to refer specifically to this implementation activity. However, we prefer to use it more broadly to encompass the overall process, which also includes modelling, design, verification, and validation.

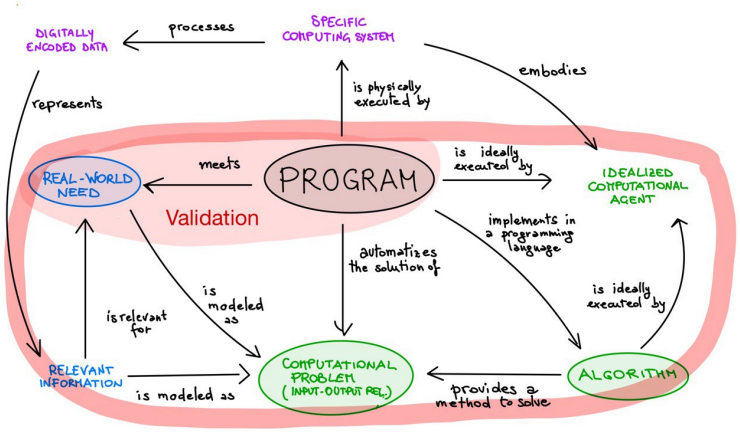


For a program to be executed by a computing system it may be necessary to translate it from the programming language it is written in, into a specific language understood by the computing system, and this is done automatically by other specialised programs (e.g., compilers). The term "program" may refer both to the program written in the programming language and its executable version. To distinguish between these two entities, the former is also called *program source*, and the latter *executable program*. The program source is in fact not only a description addressed to machines, but also a means of communication among programmers and all the stakeholders interested in understanding the details of the computation (see the notational facet).



Verification - A program should automatize the solution of the computational problem it was written for. However, the design and implementation processes are complex enterprises in themselves and not error-free; the resulting program might contain flaws (sometimes called *bugs*). The verification process aims at testing whether the program does indeed solve the computational problem, i.e., if it is able to provide the expected output for any input data.The verification process is based on the possibility of running the program on actual computing systems, on a range of different input data. To plan a comprehensive range of tests is in general a difficult task, since programs usually have to deal with a huge number of different cases. A failure during program verification may lead to further revisions of the program itself or even of its overall design.

Validation - A successful verification of a program is not enough to guarantee the appropriateness of the program from the user perspective, as the verification process refers only to the computational problem (and not to the real-world need modelled by the computational problem). In particular, decisions regarding the modelization of relevant information can have a significant impact on the quality of the overall process, and this might become evident only once the program is ready and can be executed and used. This adherence to the real-world need must be validated involving the targeted users of the program.



Example: grocery checkout

Modelling: prices are modelled as numbers; the real-world need is described computationally as the problem of summing up a series of numbers.
Design: we design an algorithm that uses a variable to keep track of the total and update it repeatedly.
Implementation: the algorithm is written in a programming language specialised for checkout devices.
Verification: the program is tested by using several series of product barcodes.
Validation: even if the program is correct in summing up for a series of products, the shopkeepers might find it not adequate because they need the program to be also able to include discounts or to compute the change.

B THE SURVEY

Our online survey was presented as a Google form. The information and questions used were as follows.

Your comments on the "The Nature of Programs" document

In the document "The Nature of Programs", we explore what a computer program is. This may seem like a trivial question, but there are so many different aspects to programming that it's valuable to explore them. We are particularly interested in doing this so that educators can develop a broad view of what the purpose of teaching programming is, and deal with a range of views they may encounter as students engage with the subject.

The document has been written by a group of Computer Science Education researchers (WG5 at ITiCSE 2022), and is available at `https://bit.ly/NoPWG5`

If you are an educator or a CS expert interested in CS education, we would be grateful to have your feedback about the scientific soundness of the document, and its interest, usefulness, and readability for educators. Please, read the accompanying document first, and then complete the survey.

Thanks you for your contribution!

Violetta & Andy (WG5 Leaders)

Privacy

The survey is anonymous.

We ask that you select your role as an "Educator" and/or "CS expert" in order to present only the relevant questions for you. The collected answers will form the basis for revising and improving the document. Professor Andrej Brodnik from University of Primorska (Slovenia) is the key contact for this project and if you require further information about this project, you can contact him at `andrej.brodnik@upr.si`

[Consent:] I consent that my answers to this survey will be analyzed and used by the members of WG5 at ITiCSE 2022 to revise and improve the document "The Nature of Programs" which will be published as a report by ACM. [A yes/no response was invited.]

Part I - Facets of programs

Please select the corresponding option for the following statements (after reading Part 1 of the document "The Nature of Programs" `https://bit.ly/NoPWG5`).

[The following statements each invited a 5-point Likert scale response from "strongly disagree" (1) to "strongly agree" (5).]

- I found the idea of characterising programs through the facets interesting.
- I found the idea of characterising programs through the facets useful.
- I found the facets to be insightful.
- I found the facets easy to understand.

[The following statements each invited a 5-point response from "strongly disagree", "disagree", "neutral", "agree", "strongly agree" for each of the six facets: Tool, Human Made, Physical Object, Abstract Entity, Executable Entity, Notational Artefact.]

DESCRIPTION - For each Facet, please select the level that matches the statement "The description of this facet makes sense". If you disagree with any of the statements, you can add a comment below.

IMPACT - For each Facet, please select the level that matches the statement "The impact section of this facet helped me better understand this facet". If you disagree with any of the statements, you can add a comment below.

EXAMPLES - For each Facet, please select the level that matches the statement "The examples given helped me better understand this facet". If you disagree with any of the statements, you can add a comment below.

[Open question:] If you disagree with any of the statements above, please elaborate on why.

[Open question:] Do you have any additional comments and/or suggestions about the multifaceted characterisation of programs?

Part 2 - How programs are created

[The following statements each invited a 5-point Likert scale response from "strongly disagree" (1) to "strongly agree" (5).]

- I found the concept map and the text describing it (page xvi) clear and easy to understand.
- I found the description of the processes involved in programming (pages xix-xxi) clear and easy to understand.
- I find the idea of describing how programs are created through the concept map interesting.
- I find the idea of describing how programs are created through the concept map useful.

[Open question:] If you disagree with any of the statements above, please elaborate on why.

For educator only

[Participants who indicated that they are an educator were given the following questions.]

Select the option that best describes your role

- Primary school teacher
- Secondary school teacher of computer science
- Secondary school teacher of a different subject
- Professor / lecturer / teaching assistant at university
- Other [specify]

Please select your level of programming experience.

- no experience
- beginner
- experienced

[The following statements each invited a 5-point Likert scale response from "strongly disagree" (1) to "strongly agree" (5).]

- I find the facets (Part 1) relevant from an educational perspective
- I find the concept map and the text describing it (page 16) relevant from an educational perspective.
- I find the description of the processes involved in programming (pages 19-21) relevant from an educational perspective.
- I find the examples in Part 2 "How programs are created" relevant from an educational perspective.

For CS expert only

[Participants who indicated that they are a CS expert (e.g., CS academic, CS practitioner, CS PhD student, professional programmer) were given the following questions.]

- Computer scientist - academic
- Computer scientist - practitioner
- Computer Science student (university level)
- Computer Science teacher at secondary school
- Other …

Are your interests/roles related to education?

- Yes
- No

[The following statements each invited a 5-point Likert scale response from "strongly disagree" (1) to "strongly agree" (5).]

- I find the multifaceted characterisation of programs (Part 1) scientifically sound.
- I find the concept map and its description (page 16) to be scientifically sound.
- I find the description of the processes involved in programming (pages 19-21) scientifically sound.

[Open question:] Did you find any factual error or inconsistencies in the document?

Conclusions

[The following open questions were given:]

- Is there anything that you feel is missing in the document?
- Is there anything in the document that you feel is misleading?
- Do you have any further comments or suggestions you would like to make?