On Abstraction and Informatics

Tom Verhoeff

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB EINDHOVEN
The Netherlands
`T.Verhoeff@tue.nl`

Abstract. One often hears, and less often reads, the claim that informatics and its application is so difficult because it involves demanding abstractions. Abstractions in informatics supposedly are even harder than those in mathematics and the physical sciences. If abstraction is so important in informatics, then you would expect that we have good ways of dealing with it and communicating about it. To some extent, this is indeed the case, but unfortunately these ways are not widespread. It is our duty to get to grips with abstraction, and especially to address it in the teaching of informatics.

In this article, I will not solve the problems posed by abstraction, and certainly not the problem of teaching abstraction. But I would like to put it more prominently on the agenda. In order to deal with abstraction, you will have to investigate it, dissect it, analyze it, establish terminology, etc. I will give a, somewhat personal, overview of abstraction, showing that it is not a single, atomic concept, but a diverse complex of interrelated concepts. It is my hope that this will help in embedding abstraction more explicitly in the informatics curriculum.

1   Introduction

Abstraction is often mentioned as explanation for why informatics[1] is so difficult. Let me quote a few of the more explicit authors. Bucci et al. start their paper [5] as follows.

> "Abstraction is one of the cornerstones of software development and is recognized as a fundamental and essential principle to be taught as early as CS1/CS2. Abstraction supposedly can enhance students' ability to reason and think. Yet we often hear complaints about the inability of CS undergraduates to do that."

In 2006, Kramer and Hazzan organized an ICSE workshop on *The Role of Abstraction in Software Engineering.* Their summary [18] explains:

---

[1] I use the term *informatics* as synonymous with *computer science.*

> "The rational for the workshop stems from the observation that some software engineers seem to be able to produce clear, elegant designs and programs, while others cannot. Abstraction is suggested as the source for this phenomenon, i.e., that some engineers lack the ability to think abstractly and to exhibit abstraction skills . . . "

Kramer later elaborated on this theme in [19]. Wing states squarely in [27]:

> "The essence of computational thinking is *abstraction*. In computing, we abstract notions beyond the physical dimensions of time and space. Our abstractions are extremely general because they are symbolic, where numeric abstractions are just a special case. . . . [O]ur abstractions tend to be richer and more complex than those in the mathematical and physical sciences."

The IEEE-ACM computing curriculum recommendations [1] also recognize the importance of abstraction in informatics. In [2, p.8], the task force writes:

> "Software engineering differs from traditional engineering because of the special nature of software, which places a greater emphasis on abstraction, modeling, information organization and representation, and the management of change. . . . An important aspect . . . is that the supporting process must be applied at multiple levels of abstraction."

Saying that abstraction is important is one thing. Explaining what abstraction is and how to teach it are quite other matters. Bucci et al. express this even more strongly in their paper [5], titled 'Do We Really Teach Abstraction?':

> "It is the central claim of this paper that, despite broad agreement about their fundamental importance, both information hiding and abstraction are severely shortchanged by current CS1/CS2 pedagogy. More specifically, we claim that standard pedagogical practices tend to compromise the human dimension of information hiding and fall well short of helping student learners to realize an increased ability to reason and think *carefully* and *rigorously* as a benefit of using abstraction."

1.1 Overview

In this article, I make an attempt at dissecting the notion of abstraction, especially, the way in which it plays a role in informatics. Section 2 considers abstraction from a philosophic-linguistic-psychologic point of view. Sections 3 and 4 address abstraction in mathematics and physics. In Section 5, I elaborate on the use of abstraction in informatics, and Section 6 focuses on the issues of teaching abstraction. Section 7 concludes this article.

Let me hasten to say that this article is not intended to serve as an example of how to incorporate abstraction into the informatics curriculum. The intention is to stimulate and organize the discussion about abstraction and its teaching.

2 Abstraction in Abstracto

Abstraction is itself a highly abstract notion; try explaining it to a six-year old child. That child can already handle some pretty abstract concepts: counting numbers, colors, relationships like brother and sister, but usually not abstract-ness itself. Let us first look at the various word usages involving abstraction, as it concerns us here (there are other meanings, not relevant for our discussion).

to abstract (intransitive verb; used with 'from X') to ignore or suppress aspects (X) that are considered irrelevant for the purpose at hand
abstraction (noun) the act of abstracting; also, the result of abstracting
abstract (adjective) the quality of being the result of abstracting; opposite of concrete

When speaking about abstraction, it is always important to ask: abstracting *from what*? For example, when counting people in a village, you '*abstract from the identity of the persons counted*'. By the way, when counting, you abstract from more than just identity; you abstract from everything but numerosity (cardinality). It is noteworthy that with this verb we typically can and do say what is suppressed (X, 'identity' in the example), but with this verb it is not so easy to say what is (intended to be) preserved ('number' in the example). You could try to do so by adding 'obtaining Y', although that does not sound so well: 'abstract from the identities obtaining (just) a number'. Also note that this verb concerns an action, transforming an initial 'concrete' 'thing' (before abstracting; like a group of people) into a resulting 'abstract' 'thing' (after abstracting; like a number).

The example of counting as a form of abstraction is relatively straightforward. Abstraction, however, also concerns the creation of new notions from existing notions, by ignoring certain features that are deemed irrelevant in the new notion. The case of counting is relatively straightforward when the notion of cardinality is already known. But what about a situation where you want to introduce a new notion, something that is not yet known? You can try to say what to ignore, but not what it is that you want to obtain, because that is precisely what is being defined. For example, how to define the notion of *probability* when this notion did not yet exist? I will get back to this in Section 3, when dealing with abstraction in mathematics.

Note that the invention of new notions is a creative act, and an important aspect of doing science (also see Section 4 on physics). Epistemology is the branch of philosophy concerned with knowledge and its creation. But the creation of new notions is important in teaching too, because people are not born with the latest notions from science embedded in their brains. The situation of a student resembles that of the initial inventors of those concepts. So, the process of creating new notions is not a one-time thing, but is repeated every time a person tries to understand an unfamiliar notion. Since abstraction is one of the most effective ways to define new notions, teachers should be well aware of this mechanism, so that they can properly guide their students.

Why is abstraction needed at all? The main reason derives from cognitive limitations of the human mind. We, human beings, have very limited processing powers. Our short-term memory can store only a few 'chunks' of information at a time. Traditionally, this limit was believed to be 'seven plus or minus two' [24], but recent research has decreased that limit to approximately four [6]. By applying appropriate abstractions, we can concentrate on relevant information and ignore everything else. Without abstraction, our minds would be overwhelmed with information and we would not be able to function as effectively as we do.

It is good to point out some misconceptions about abstraction. Abstraction is not the same as *vagueness* or *imprecision*, although these also concern ways of limiting information content. Unfortunately, half-baked attempts at teaching abstraction get no further than just vague and imprecise bluffing.

Dijkstra expresses the need and the misconception crisply in his Turing Award Lecture [13]:

> "We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad cases is called "abstraction"; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer. In this connection it might be worth-while to point out that the purpose of abstracting is not to be *vague*, but to create a new semantic level in which one can be absolutely precise."

Another misconception is that abstract–concrete is a dichotomy, a black-and-white thing. In linguistics this seems to be the case, where you have concrete entities (people, rocks, paper, scissors) and abstract notions (number, color, relationships). But in scientific usage, and especially in informatics, abstraction concerns a gradual scale. You can abstract from several aspects, one by one, in separate abstraction steps, yielding a sequence of intermediate concepts of increasing abstraction. Thus, abstract–concrete is a *relative* relationship: B can be more abstract than C, and A can in turn be more abstract than B. This gives rise to multiple levels or layers of abstraction.

All of this shows that abstraction is not an easy concept, both from a linguistic and a philosophic point of view. Still, that is no reason to shy away from it. On the contrary, it poses an important challenge to be tackled.

3 Abstraction in Mathematics

Mathematics is a good place to start digging into the notion of abstraction, because it is itself abstract (a purely mental construction), and a rigorous discipline, which developed precise ways of defining abstractions. As such, it is also an important foundation for informatics. Devlin argues this in [12]:

> "The main benefit of learning and doing mathematics is that it develops the ability to reason about formally defined abstract structures like those in computer science and its applications."

How are mathematical concepts defined? Basically, there are only two methods:

- axiomatically, by postulating a collection of defining properties
- in terms of existing concepts, possibly through abstraction

I do not want to get sidetracked into the philosophy of mathematics, more particularly, into the ontology of mathematical objects. So, the following presentation is necessarily simplified. A well-known example of the *axiomatic method* is the introduction of the concepts of Euclidean geometry. The definitions of *point*, *line*, and *incidence* do not tell what these concepts *are* in themselves. Instead, they postulate relationships, that together sufficiently define those concepts to do mathematics, that is, to formulate and prove interesting theorems. For instance, Euclid's first axiom states that two distinct points determine a unique line incident on these points. Similarly, the *Dedekind–Peano Axioms* define the natural (counting) numbers, and the *Zermelo–Fraenkel Axioms* define sets. The biggest danger of axiomatic definitions is *inconsistency*, i.e., that the concepts intended to be defined actually cannot exist, because the axioms contradict each other. This is typically a hard problem, or even impossible to resolve.

In the second method, new concepts are defined in terms of existing concepts. Therefore, such definitions are more easily shown to be valid (consistent). For instance, a (mathematical) *graph* is defined as a pair (V, E) where V is a set (whose elements are called *vertices*) and $E \subseteq V \times V$ is a set of vertex pairs (called *edges*).

3.1 Definition by Abstraction

An important variant of this second method is the *definition by (logical) abstraction*. The *integer numbers* can be defined as 'signed' natural numbers, i.e., as pairs of a plus or a minus sign, and a natural number, where plus zero and minus zero are the same thing. More elegant, however, is the following definition involving the set $P = N \times N$ (pairs of natural numbers) and the relation $\sim$ on P defined by

$$(a, b) \sim (c, d) \iff a + d = b + c \tag{1}$$

This definition is expressed in terms of existing concepts, viz. addition and equality of natural numbers and well defined for all a, b, c, and d. If you already know the integer numbers, then I can reveal to you that the pair $(a, b) \in P$ is intended to 'correspond' to the integer $a - b$ (like the way bookkeepers avoid negative numbers by working in two columns). Note, however, that $a - b$ is not (yet) defined for natural numbers a and b when $a < b$. Relation $(a, b) \sim (c, d)$ intends to capture $a - b = c - d$, which is equivalent to the right-hand side of (1). But you do not need to know all that to read the definition.

It turns out that relation $\sim$ is an *equivalence relation*, i.e., it is *reflexive* ($p \sim p$), *symmetric* ($p \sim q$ implies $q \sim p$), and *transitive* ($p \sim q$ and $q \sim r$ implies $p \sim r$). An equivalence relation on a set P partitions P into disjoint nonempty subsets, such that equivalent elements are in the same part, and non-equivalent

elements are in distinct parts. The set Z of integer numbers is now defined as the set of *equivalence classes* in P under $\sim$. That is, each equivalence class, i.e., set of $\sim$-equivalent pairs of natural numbers, is an integer number in Z. This is also written as $Z = P/\sim$, the 'quotient' of P and $\sim$. The equivalence relation is 'divided out'; the distinctions between equivalent pairs are 'erased' by uniting them into a single new 'concept', called integer number. The integers are, thus, obtained as pairs of natural numbers where we abstract from the distinction between pairs satisfying (1).

At first sight, this looks like a horribly complicated definition, because an integer now 'is' a set of $\sim$-equivalent pairs of natural numbers. For these integers, one can define operations like addition, subtraction, and multiplication, and prove their basic properties. Further analysis shows that every pair in P is $\sim$-equivalent either to a pair of the form $(a, 0)$, which can be identified with the natural number a, or to a pair of the form $(0, a)$ with $a > 0$, which is usually written as $-a$. Once, this is done, you no longer need to worry about the fact that an integer was defined as such a horribly complicated object. You can simply use the properties, and think of integers as 'atomic' objects. The quotient construction merely proves the consistency of the properties. It is an 'internal' concern for foundationalists that need not concern mere users of the integers.

Once one is familiar with the method of definition by abstraction, it becomes a powerful tool. In fact, every abstraction can be can be viewed as abstracting from an appropriate equivalence relation. For instance, the *rational numbers* can be defined by abstracting from the equivalence relation $\sim'$ defined by

$$(a, b) \sim' (c, d) \Longleftrightarrow a \times d = b \times c \tag{2}$$

on pairs of integer numbers, where the second element is nonzero. Defined in this way, rational numbers involve two layers of abstraction based on equivalence relations (1) and (2).

3.2   How to Work with Definitions

How one handles a definition, depends on its nature. Axiomatic definitions are eliminated not by applying them to a single concept, such as a geometric point, but by applying them to a suitable *combination* of concepts that are related through an axiom, such as two distinct points that yield a line incident on those points.

Definitions in terms of existing concepts can be eliminated by *substitution*. For instance, the number $M(a, b)$ halfway between numbers a and b can be defined as $(a + b)/2$. When you want to prove something about $M(x + y, x - y)$, you can simply eliminate M by substitution, obtaining $((x + y) + (x - y))/2$, which incidentally can be further reduced to x.

There is one exception to this elimination: *recursive definitions*. These involve a new mechanism, where something is, partly, defined in terms of itself. For instance, $n!$ (n factorial) can be defined for natural numbers $n \geq 0$ by

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n \geq 1 \end{cases} \tag{3}$$

When you want to prove something about an expression of the form $(a + b)!$, you cannot simply eliminate ! by substitution. You need to resort to *induction*. In teaching, the notions of recursion and induction are always challenging [29].

Category Theory can be viewed as the summum of abstraction in mathematics. It is a theory about mathematical structure in very general terms, with *functions* as fundamental notion. I am not saying that you need to study Category Theory before teaching about abstraction. But it does serve to illustrate how the suppression of internal details makes for new ways of defining things. The traditional definition of an injective function (injection or one-to-one function) is as follows. Function $f : A \rightarrow B$ is called *injective* when

$$f(x) = f(y) \text{ implies } x = y, \text{ for all } x, y \in A \tag{4}$$

This definition involves the application of the function to arguments, thereby relying on 'internal details' of functions. In Category Theory, functions are usually referred to as *morphisms*, and an injection is called a *monomorphism*. Morphism $f : A \rightarrow B$ is called a monomorphism when

$$f \circ g = f \circ h \text{ implies } g = h, \text{ for all } g, h : Z \rightarrow A \tag{5}$$

where $\circ$ denotes morphism composition; in traditional terms: $(f \circ g)(x) = f(g(x))$. Definition (5) is *point-free*, i.e., only relies on 'external features' of morphisms, without applying them to specific arguments. The definition of an *epimorphism* (surjective function) is a completely analogous dual of (5):

$$g \circ f = h \circ f \text{ implies } g = h, \text{ for all } g, h : B \rightarrow Z \tag{6}$$

Compare this to the traditional definition, which looks very different from (4):

$$(\forall\, y \in B : (\exists\, x \in A : f(x) = y)) \tag{7}$$

Thus, (5) and (6) are more abstract than (4) and (7). This also relates to functional programming (see Section 6).

On one hand, mathematical concepts are abstract and people often express having difficulty with mathematics. On the other hand, recent research [10,25] has revealed that the human mind is born with powerful innate abstraction abilities. For instance, even newly born babies already have an innate ability to handle small numbers 'in the abstract'. This partly overthrows the traditional cognitive development sequence as described by Piaget, where abstract thinking would only appear in the formal operational stage at the age of twelve or so.

In [11], Devlin makes the case that the human brain has evolved to deal with precisely those abstract structures that are mathematically important. His argument runs as follows. Human beings are relatively weak, both on attack and defense, compared to most (other) animals. In order to survive, they have had to evolve brain structures to cooperate in larger groups. Such cooperation requires that the brain keeps track of relationships among people in groups. It is precisely such networks of relationships that are important in mathematics as well. All human brains are basically equipped to deal with mathematical structures. Why not everybody is a math whiz, is another story (see [11]).

4 Abstraction in Physics

Abstraction in the physical sciences serves a similar purpose as in mathematics, viz. to define concepts. These concepts are used to describe (laws of) nature. They are abstractions created by the human mind. Think of concepts like mass, force, energy, and temperature. In contrast to mathematics, such concepts are intended to have a link to the concrete physical world. Here we have a clear jump from concrete to abstract. After abstraction, we end up in the mathematical world.

One particularly important abstraction step in physics concerns the way a system's behavior over time is modeled. A dynamical system is difficult to analyze when you try to 'think it through', tracing all changes over time. Your mind is soon overwhelmed. The key idea is to capture such behavior by considering the system's state *as a function of time*. This function is a single entity, containing all information that is relevant. Such functions can be handled at a higher level of abstraction. For instance, a law can be expressed as a differential equation involving the system's state function. All of time is handled in one swoop.

Under the heading "The Simplification of Science and the Science of Simplification" in [26, pp.8–12], Weinberg discusses Newton's Law of Universal Gravitation (by Feynman called "the greatest generalization achieved by the human mind"). This law expresses the attractive force between two bodies as function of their mass and distance. First of all, he notes that this law implicitly says that the attractive force does not depend on anything other than the two masses and their distance. One can abstract from everything else, a huge abstraction. When Newton used his law to analyze our solar system, he had to apply further abstractions. In particular, to calculate a planet's orbit, he focused on the sun and that planet only, ignoring all other masses and (non-gravitational) influences. Weinberg writes:

> "Newton was a genius, but not because of the superior computational power of his brain. Newton's genius was, on the contrary, his ability to simplify, idealize, and streamline the world so that it became, in some measure, tractable to the brains of perfectly ordinary men."

5 Abstraction in Informatics

In informatics and its application, both the mathematical and the physical kind of abstractions play a role. When automating the processes in a car, the software will somehow have to be related to various (physical) car parts. In the software, these parts exist as abstractions. This kind of physical abstraction is encountered when modeling the problem domain. As with physics, information processing systems involve behavior over time, where the notions of state and state change in a state space are important. Also see [9], [16, Ch.1], [3], and [28].

The definition of models, both for the problem domain and the solution domain, involve the mathematical kind of abstractions. The main abstractions in theoretical informatics concern the notions of *algorithm* and *information*.

The reason that abstraction plays a more important role in informatics than in other disciplines, is that the description of every program involves the creation of new abstractions to define the data and the operations. To aid the definition of data and operations, many predefined generic abstractions are available, and must have been mastered.

Programming tools offer various layers of abstractions on top of the hardware that executes the programs. Compilers, interpreters, operating systems, and run-time libraries translate high-level descriptions into low-level instructions. Furthermore, modern programming languages provide abstraction mechanisms to aid in the description of data structures and algorithms. Let us briefly look at some of these abstraction mechanisms (see e.g. [20]):

procedural abstraction: mechanism to define parameterized routines (functions, procedures, methods) usable as an action

data abstraction: mechanism to define parameterized (generic) data types (sets of values with associated operations)

iteration abstraction: mechanism to iterate over a collection, processing each element once; there are also many loop patterns

Note that non-recursive routine and data type definitions can be eliminated by a double substitution: substitute actual for formal parameters, and substitute the body or representation for the invocation or occurrence.

These mechanisms share a number of features:

- abstraction from operational and representational implementation details, as seen from the user's perspective;
- abstraction from the identity (in case of procedural abstraction) or type (in case of data abstraction) of data involved, via parameters;
- abstraction from context of usage[2], as seen from the provider's perspective.

They involve two sides: a *user* or *client* side and a *provider* or *server* side. These sides are completely separate, except for a two-sided contract that specifies the defined entity, thereby serving as an *interface*. In reasoning about these abstractions, it is mandatory (for otherwise the abstraction loses its value) to avoid reasoning that relates the user context and the provider context directly to each other (basically, by carrying out the substitution mentioned above). Instead, both the user context and the provider context should solely be related to the agreed contract. For instance, in the case of routines, the contract is expressed in terms of a *precondition* and a *postcondition* (also see Fig.1):

- The user takes care that the precondition holds.
- The provider exploits this precondition and takes care of the postcondition.
- The user exploits this postcondition.
- The user need not know how the routine does its work.
- The provider need not know how the parameters and result are used.

---

[2] This abstraction is often not mentioned, but equally important as the first.

	Two-sided Contract	
	Precondition	**Postcondition**
User	concern	benefit
Party	↓	↑
Provider	benefit →	concern

Fig. 1. The relationships in two-sided contracts for routines

We see here that abstraction appears in many disguises, going by such names as *divide and conquer, separation of concerns, modularization, generalization, Design By Contract, encapsulation, information hiding, modeling and design patterns*, etc. These techniques aim at managing complexity through abstraction.

In software development, the path from specification to implementation involves multiple refinement steps, going from abstract models to more concrete code. In the analysis of existing systems, models need to be constructed for those systems, thus going from concrete to more abstract.

6 Teaching Abstraction

Many textbooks present various abstraction mechanisms in programming languages, but most do so implicitly. One of the more explicit textbooks is [20]. Others textbooks present modeling and design techniques, again with limited attention for abstraction. The pedagogy of abstraction in informatics has not been studied deeply. Useful articles are [5,14,17,19]. The teaching of abstraction must be based on a long-term plan. Abstraction cannot be taught in a few lessons; it must be infused over many years. That requires a carefully tuned curriculum. Offering curricular advice is beyond the scope of this article. In summary, [17] advises to start early; to teach abstraction consciously; to stress the benefits of using abstraction. I would like to add: teach it in small steps.

The question remains: How to do so? Consider the case of teaching abstraction mechanisms like parameterized procedures and Abstract Data Types (ADTs). In a traditional approach, you teach such mechanisms by explaining and practicing the steps involved in the order of the development process; see Fig. 2 (left). The student then always works toward the unknown. An alternative approach is *backward chaining*, which I learned from [8, p.38] (also see [4]). Here, a process consisting of several steps is taught in reverse order, by starting with the *last* step, and successively adding *preceding* steps, one by one; see Fig. 2 (right). In backward chaining, the student always works toward familiar steps. Backward chaining is also applied successfully in sports, music, and military training. It resembles Meyer's *inverted curriculum* and *outside-in method* [21,23]. It would be interesting to see whether abstraction in informatics can be captured in a set of Meyer's TRUCs [22], i.e, testable and reusable units of cognition.

Unfortunately, many modern programming languages, like C, C++, C#, and Java do not provide sufficient support for abstraction. For instance, only the syntactic part of contracts can be expressed as part of the program, while the

In the development process, you	In backward-chaining, you learn to
(1) Draw up a contract for an abstraction	(1) Use an abstraction, given its contract
(2) Validate the contract	(2) Test an abstraction, given its contract
(3) Design/implement it (can be split)	(3) Review a given design/implementation
(4) Review the design/implementation	(4) Design/implement a given contract
(5) Test the implementation	(5) Validate contracts
(6) Use the abstraction	(6) Draw up a contract for an abstraction

Fig. 2. Forward and backward chaining

semantic part must be expressed as comment. The contract for a function cannot be named separate from its implementation. There are some add-on features for these languages, but they are not appropriate for teaching. Eiffel [23], Perfect Developer [7], and RESOLVE [5] are a considerable step in the right direction.

The role of efficient and effective notation should not be underestimated [15, Ch.16]. In this respect, *functional* and *declarative* languages are better alternatives than *imperative* languages. But they are —unrightfully— too often avoided in teaching, probably because of the higher level of abstraction. I agree with [5], in that we need to teach students about abstract mathematical notions, like sets, mappings, relations, bags, and sequences, before they start to develop implementations. They need these notions to read proper contracts and to express models without thinking in terms of implementations. But it is also important that model-oriented specifications are complemented by property-oriented specifications (resembling the axiomatic definitions in mathematics). In particular, reasoning about abstractions is underexposed.

7 Conclusion

We have looked at abstraction from various angles, especially as it plays a role in informatics. It is important that abstraction is properly integrated into informatics teaching. Our hope is that this overview will organize and stimulate the discussion on how to teach abstraction in the informatics curriculum.

References

1. ACM–IEEE Joint Task Force on Computing Curricula. *Computing Curricula 2001 Computer Science: Final Report*. Dec. 2001.
   `www.acm.org/education/curric_vols/cc2001.pdf` (accessed Apr. 2011)
2. ACM–IEEE Joint Task Force on Computing Curricula. *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*. Aug. 2004. `sites.computer.org/ccse/SE2004Volume.pdf` (accessed Apr. 2011)
3. Dines Bjørner. *Software Engineering 1: Abstraction and Modelling*. Springer, 2006.
4. B. Brandon. "Last Things First: The Power of Backward Chaining", *The eLearning Developers' Journal*, Oct. 2003.
5. P. Bucci, T. J. Long, B. W. Weide. "Do We Really Teach Abstraction?", *SIGCSE Bull.*, **33**(1):26–30 (Feb. 2001).

6. Nelson Cowan. "The Magical Number 4 in Short-Term Memory: A Reconsideration of Mental Storage Capacity", *Behavioral and Brain Sciences*, **24**(1):87-114 (2001).
7. D. Crocker, J. Carlton. "Perfect Developer: what it is and what it does", *FACS Facts*, newsletter of the BCS Formal Aspects of CS special interest group, Nov. 2004. `www.eschertech.com/papers/introduction_to_perfect_developer.pdf` (accessed Apr. 2011)
8. E. de Bono. *Teach Your Child How to Think*. Penguin, 1994.
9. O.-J. Dahl, E. Dijkstra, C. Hoare, *Structured Programming*, Academic Press, 1972.
10. S. Dehaene. *The Number Sense: How the Mind Creates Mathematics*. Oxford Univ. Press, 1997.
11. K. Devlin. *The Math Gene: How Mathematical Thinking Evolved and Why Numbers Are Like Gossip*. Basic Books, 2000.
12. K. Devlin. "Why universities require computer science students to take math", *Communications of the ACM*, **46**(9):36–39 (Sep. 2003).
13. E. W. Dijkstra. "The Humble Programmer" (Turing Award Lecture 1972). `www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD340.html` (accessed Apr. 2011)
14. P. Frorer, O. Hazzan, M. Manes. "Revealing the Faces of Abstraction", *International J. Computers for Mathematical Learning*. Kluwer Academic Publishers, **2**(3):217–228 (Oct. 1997).
15. A. J. M. van Gasteren. *On the shape of mathematical arguments*. LNCS **445**, Springer, 1990.
16. Hans Jonkers. *Abstraction, Specification and implementation techniques: with an application to garbage collection*. Dissertation, Eindhoven Univ. of Technology, 1983.
17. H. Koppelman, B. van Dijk. "Teaching abstraction in introductory courses", *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education ITiCSE '10*, pp.174–178 (2010).
18. J. Kramer, O. Hazzan. "Summary of an ICSE 2006 Workshop: The Role of Abstraction in Software Engineering", *ACM SIGSOFT Software Engineering Notes*, **31**(6):37–42 (Nov. 2006).
19. J. Kramer. "Is Abstraction the Key to Computing?" *CACM* **50**(4):36–42 (Apr. 2007).
20. B. Liskov, J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.
21. Bertrand Meyer. "The Outside-in Method of Teaching Introductory Programming." In M. Broy and A. V. Zamulin (Eds.), *Ershov Memorial Conference*, LNCS **2890**:pp.66–78. Springer, 2003.
22. B. Meyer. "Testable, Reusable Units of Cognition", *IEEE Computer*, **3**(4):20–24 (Apr. 2006)
23. B. Meyer. *Touch of Class: Learning to Program Well with Objects and Contracts*. Springer, 2009.
24. George A. Miller. "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information", *Psychological Review*, **63**:81-97 (1956).
25. D. A. Sousa. *How the Brain Learns Mathematics*. Corwin Press, 2008.
26. G. M. Weinberg. *An Introduction to General Systems Thinking*. Wiley, 1975.
27. J. Wing. "Computational thinking and thinking about computing", *Phil. Trans. R. Soc. A*, **366**:3717–3725 (Jul. 2008).
28. Niklaus Wirth. "A Brief History of Software Engineering", *IEEE Annals of the History of Computing*, **30**(3)32–39 (July–Sept. 2008).
29. D. R. Woodal. "Inductio ad Absurdum?", *Mathematical Gazette*, **59**(408):64–70 (Jun. 1975).