# IS ABSTRACTION THE KEY TO COMPUTING?

*Why is it that some software engineers and computer scientists are able to produce clear, elegant designs and programs, while others cannot? Is it possible to improve these skills through education and training? Critical to these questions is the notion of abstraction.*

*By* JEFF KRAMER

For over 30 years, I have been involved in teaching and research in computer science and software engineering. My teaching experience ranges from courses in programming, to distributed systems, distributed algorithms, concurrency, and software design. All these courses require that students are able to perform problem solving, conceptualization, modeling, and analysis. My experience is that the better students are clearly able to handle complexity and to produce elegant models and designs. The same students are also able to cope with the complexities of distributed algorithms, the applicability of various modeling notations, and other subtle issues.

On the other hand, there are a number of students who are not so able. They tend to find distributed algorithms very difficult, do not appreciate the utility of modeling, find it difficult to identify what is important in a problem, and produce convoluted solutions that replicate the problem complexities. Why? What is it that makes the good students so able? What is lacking in the weaker ones? Is it some aspect of intelligence? I believe the key lies in abstraction: The ability to perform abstract thinking and to exhibit abstraction skills.

The rest of this article explores this hypothesis and makes recommendations for future work. We first discuss what abstraction is and its role in computing and other disciplines. Others, such as Hazzan [5] and Devlin [2] have also discussed abstraction as a core skill in mathematics and computing. Using findings from cognitive development, we explore the factors affecting students' ability to cope with and perform abstraction. We discuss whether or not abstraction is teachable and suggest that steps must be taken to test abstraction skills as a means of validating our hypothesis, checking our teaching techniques and even, perhaps, selecting our students.


**Figure 1. Henri Matisse, "Naked Blue IV," 1952; paper cutouts.**

## Abstraction: What Is It? Why Is It So Important?

From the definitions of abstraction [10], we focus on two particularly pertinent aspects.[1] The first emphasizes the process of removing detail to simplify and focus attention based on the definitions:

- The act of withdrawing or removing something, and;
- The act or process of leaving out of consideration one or more properties of a complex object so as to attend to others.

The second emphasizes the process of generalization to identify the common core or essence based on the definitions:

- The process of formulating general concepts by abstracting common properties of instances, and;
- A general concept formed by extracting common features from specific examples.

Abstraction is widely used in other disciplines such as art and music. For instance, as shown in Figure 1,

[1] See [3] for an interesting discussion on the faces of abstraction.

Henri Matisse manages to clearly represent the essence of his subject, a naked woman, using only simple lines or cutouts. His representation removes all detail yet conveys much. In his painting "South Wind, Clear Skies" (Red Fuji), Katsushika Hokusai captures Mount Fuji (see Figure 2). Art critic (Gabrielle Farran) remarks that he uses a "perfect balance of colour and composition rendering an abstract form of the mountain to capture its essence." Another example is from jazz, where musicians identify the essential melody or heart of the particular piece of music, and improvise around that to provide their own embellishments. One jazz musician provided the following apposite remark, "It is easy to make something simple sound complex, however it is more difficult to make something complex sound simple." This difficulty is a clear example of the challenge in the application of abstraction in removing extraneous detail.

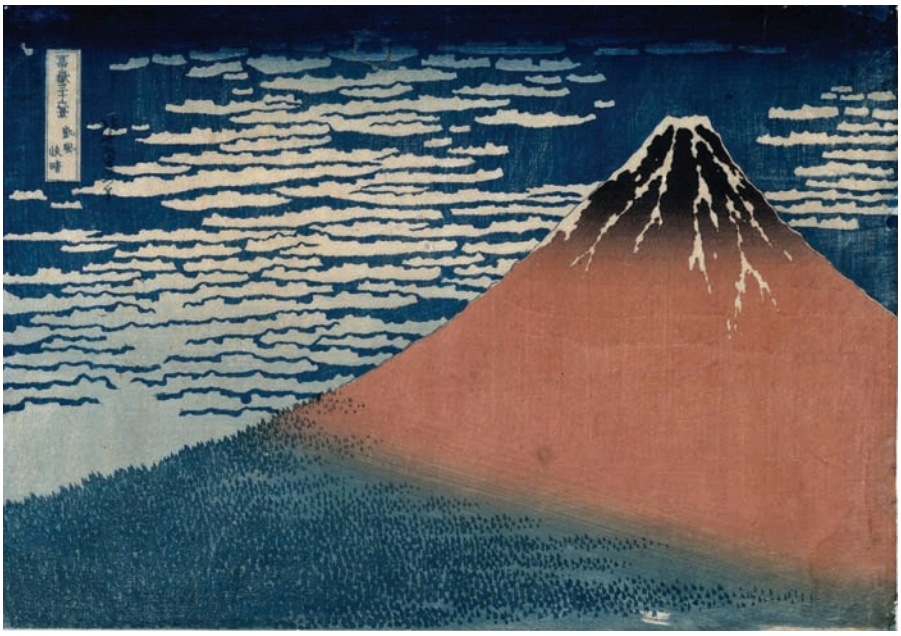

**Figure 2. Katsushika Hokusai, "South Wind, Clear Sky (Gaifʻ kaisei) or "Red Fuji," a color woodblock print, Japan, AD 1830–33.**

A wonderful example of the utility of abstraction is provided by the contribution of Harry Beck to the renowned London Underground map. In 1928, the map was essentially an overlay of the underground system onto a conventional geographical map of London (see Figure 3a). It showed the curves of the train lines and of the River Thames, and the relative distances between the stations. In 1931, Beck produced the first abstract, schematic representation, simplifying the curves to just horizontal, vertical and diagonal lines and where the distances between stations were no longer proportional to the geographical distances (see Figure 3b).

This form of simplified representation or abstraction is ideally fit for the purpose of navigating around the London Underground. Not only is this transit abstraction still used today, it has been adopted by transportation agencies in numerous countries. Indeed, the level of abstraction had to be carefully selected so as to include only the required details but neglect the unnecessary—too abstract and the map would not provide sufficient information for the purpose; too detailed and the map becomes confusing and less comprehensible. Like any abstraction, it can be misleading if used for other purposes. The underground map is sometimes misused by tourists who misinterpret it as an actual geographical map of London. The level, benefit, and value of a particular abstraction depend on its purpose.

Why is abstraction important in computer science and software engineering? Software itself is certainly abstract, and the discipline of producing software requires abstraction skills. Keith Devlin [2] states the case clearly and concisely, "Once you realize that computing is all about constructing, manipulating, and reasoning about abstractions, it becomes clear that an important prerequisite for writing (good) computer programs is the ability to handle abstractions in a precise manner." Wing [11] confirms the importance of abstraction in computational thinking by emphasizing the need to think at multiple levels of abstraction. Ghezzi et al. [4] identify abstraction as one of the fundamental principles of software engineering in order to master complexity. The removal of unnecessary detail is obvious in requirements engineering and software design.

Requirements elicitation involves identifying the critical aspects of the environment and the required system while neglecting the irrelevant. Design requires that one avoid unnecessary implementation constraints. For instance, in compiler design, one often employs an abstract syntax to focus on the essential features of the language constructs, and designs the compiler to produce intermediate code for an idealized abstract machine to retain flexibility and avoid unnecessary machine dependence. The generalization aspect of abstraction can be clearly seen in programming with the use of data abstractions and

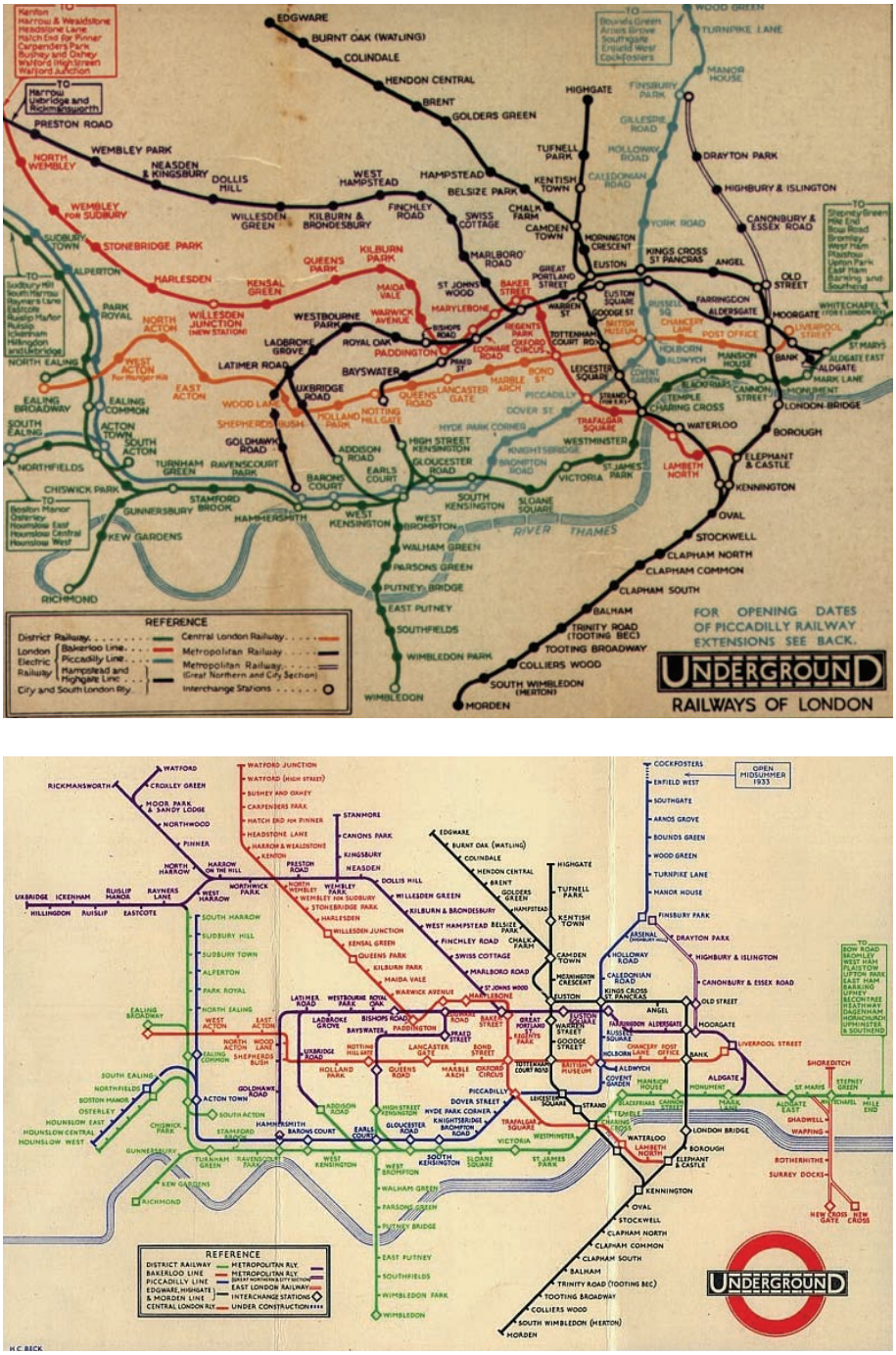


**Figure 3. The London Underground Map (a) the 1928 map and (b) the 1933 map by Harry Beck.**

classes in object-oriented programming. Abstract interpretation for program analysis is another example of generalization, where the concrete program domain is mapped to an abstract domain to capture the semantics of the computation for program analysis.

Abstraction skills are essential in the construction of appropriate models, designs, and implementations that are fit for the particular purpose at hand. Abstract thinking is essential for manipulating and reasoning about abstractions, be they formal models for analysis or programs in a programming language.

In fact, abstraction is fundamental to mathematics and engineering in general, playing a critical part in the production of models for analysis and in the production of sound engineering solutions.

### What Determines Our Students' Abilities?

Do our students' powers of abstraction depend on their cognitive development? Can we improve their abilities and, if so, how? Is it possible to teach abstract thinking and abstraction skills?

Jean Piaget (1896–1980) provided the foundations for an understanding of the cognitive development of children from infants to adulthood [6, 9]. Based on case studies, he derived four stages for development: sensorimotor, pre-operational, concrete operational, and formal operational. The first two stages are from infancy to early childhood (about the age of seven), where intelligence is roughly indicated by motor activity and then by language and early symbol manipulation respectively. The third is the concrete operational stage, from about seven to 12, where intelligence is roughly indicated by a grasp of conservation of matter, of causality and an ability for classification of concrete objects. The fourth is the formal operational stage, from around 12 to adulthood, where individuals indicate an ability to think abstractly, systematically, and hypothetically, and to use symbols related to abstract concepts. This is the crucial stage at which individuals are capable of thinking abstractly and scientifically.

Although there is some criticism concerning the way Piaget conducted his research and derived his

theory, there is general support for his underlying ideas. Further studies and experimental evidence supports Piaget's hypothesis that children progress through the first three stages of development; however it appears that not all adolescents progress to the formal operations stage as they mature. Biological development may be a prerequisite, but tests conducted on adolescent and adult populations indicate that only 30% to 35% of adolescents achieve the formal operations stage, that some adults never do [7], and that particular environmental conditions and training may be required for adolescents and adults to achieve this stage.

### Is Abstraction Teachable?

Although the low attainment figures for Piaget's formal operations stage may be rather disappointing, there does seem to be some hope of improving students' achievement by creating the right educational environment. For instance, for adolescents Huitt and Hummel [6] (based on Woolfolk and McCune-Nicolich [12]) recommend using teaching techniques such as giving students the opportunity to explore many hypothetical questions—encouraging students to explain how they solve problems—and teaching broad concepts in preference to just facts.

What about course content and curricula? At Imperial College, the four-year Masters of Engineering degree in computing offers over 60 different course modules, including a number of optional specialization courses in the third and fourth years. None of these courses is a course on abstraction, yet all rely on or utilize abstraction to explain, model, specify, reason or solve problems! This seems to confirm that abstraction is an essential aspect of computing, but that it must be taught indirectly through other topics.

Our anecdotal experience is that mathematics is an excellent vehicle for teaching abstract thinking. In our early years, when there was less mathematical content in our curricula for undergraduate courses, the students appeared to lack abstraction skills and were less able to deal with complex problems. Devlin confirms this experience by remarking, "The main benefit of learning and doing mathematics is not the specific content; rather it's the fact that it develops the ability to reason precisely and analytically about formally defined abstract structures" [2]. More detailed supporting arguments are provided by Devlin and others in *Communications*' special section "Why Students Need Math" [2]. The case in favor of a mathematical treatment of computing and the inclusion of mathematical topics in the curriculum is strong. However, in computing, it is crucial students are not only capable of manipulating symbolic and numerical formalisms, but also skilled at moving from an informal and complicated real world to a simplified abstract model.

The ACM/IEEE Computing Curricula: Software Engineering Volume 2004 [1] gives some recognition of the importance of abstraction by including aspects such as encapsulation, levels of abstraction, generalization and class abstractions; however, it is software modeling and analysis that receives major attention.

Formal modeling and analysis is a powerful means for practicing abstract thinking and consolidating students' ability to apply abstraction. Modeling is the most important engineering technique; models help us to understand and analyze large and complex problems. Since models are a simplification of reality intended to promote understanding and reasoning, students must exercise all their abstraction skills to construct models that are fit for purpose. They must also be capable of mapping between reality and the abstraction, so as to appreciate the limitations of the abstraction and to interpret the implications of model analysis.

Student motivation can be enhanced by presenting the mathematics of the modeling formalism in a problem-oriented manner, and can benefit by the provision of tool support (such as model checking) for reasoning and analysis.

My personal experience teaching model building and analysis as part of a course on concurrency [8] has been very encouraging. Given a model, students find it very helpful in clarifying the important aspects of the problem and in using a model checking tool to reason about its properties and behavior. However, some still seem to find it extremely difficult to construct the models themselves. It is not enough to think about what they want to model, they need to think about how they are going to use that model. What is the purpose of the model? Though capable of abstract thinking and reasoning, these students seem to lack the skills to apply abstraction.

### What Do We Need To Do?

If abstraction is a key skill for computing, we should focus more directly on ensuring that our teaching is effective and that computing professionals have adequate abstraction skills.

What has been presented here is mostly anecdotal, with some supporting evidence from the literature. How can we put this on a more scientific footing and improve our understanding of the situation? As in all scientific and engineering endeavors, before we can control or effect, we must first measure. The aim is to gather the following data:

*Measure students abstraction abilities annually while at college.*

This could be used to check whether or not their ability correlates with their grades, relative to others in their year. Assuming that our conventional grading techniques—coursework, laboratory work and examinations—are indicative of a student's ability in computing, this would help to gain confidence that abstraction is a key indicator of ability. A second purpose of such testing is that it would provide an alternative means for checking students' abilities. Finally, it could also help to assess the efficacy of our teaching techniques, ensuring that students of all abilities do improve as they progress through the degree course.

*Measure students abstraction abilities at the time of application to study computing.*

Currently, entry is based almost solely on school grades. Abstraction ability could potentially be used to help eliminate those students that are not suitable or less likely to perform well, and to select those who are not just academically capable, but that have a real aptitude for computing and software engineering.

Conducting these experiments and collecting this data depends on the availability of good abstraction tests for measuring students' abstract thinking and abstraction skills. Unfortunately, we have been unable to find any existing appropriate tests. Tests for the formal operations stage focus mainly on logical reasoning and are not appropriate for testing abstraction skills nor capable of distinguishing between the abilities of students at college level. Orit Hazzan, of the Department of Education in Science and Technology at the Technion, has recommended that a specific set of test questions be constructed, including sufficiently different kinds of tasks and descriptions, supporting the collection of both quantitative and qualitative data, and including open-ended questions and interviews.

These tests should examine different forms of abstraction, different levels of abstraction and different purposes for those abstractions. This must be our next step. Only then can we be more definitive as to the criticality of abstraction in computing and of our ability to teach it. For instance, we should be able to confirm or refute that a particular course, such as formal modeling and analysis, is indeed an effective means for teaching abstraction.

## Conclusion

Like others, I believe that abstraction is a key skill for computing. It is essential during requirements engineering to elicit the critical aspects of the environment and required system while neglecting the unimportant. At design time, we must articulate the software architecture and component functionalities that satisfy functional and non-functional requirements while avoiding unnecessary implementation constraints. Even at the implementation stage we use data abstraction and classes so as to generalize solutions.

This article proposes the reason that some software engineers and computer scientists are able to produce clear, elegant designs and programs, while others cannot, is attributable to their abstraction skills. I argue that a sound understanding of the concept of abstraction and its importance in software engineering, and the means to teach and to test abstraction skills are crucial to the future of our profession. The primary need is for a set of abstraction Tests for checking student progress, checking our teaching techniques, and potentially as an aid for student admissions selection. C

## References


1. The ACM/IEEE Computing Curricula: Software Engineering (2004); www.computer.org/education/cc2001.
2. Devlin, K. Why universities require computer science students to take math, *Comm of ACM 46*, 9 (Sept. 2003), 37–39.
3. Frorer, P., Hazzan, O. and Manes, M. Revealing the faces of abstraction. *International J. Computers for Mathematical Learning.* Kluwer Academic Publishers, 1997, 217–228.
4. Ghezzi, C., Jazayeri, M. and Mandrioli, D. *Fundamentals of Software Engineering, 2nd Edition.* Pearson International, NJ, 2003.
5. Hazzan, O. Reducing abstraction level when learning abstract algebra concepts. *Educational Studies in Mathematics 40*. Kluwer Academic Publishers, 1999, 71–90.
6. Huitt, W. and Hummel, J. Piaget's theory of cognitive development. *Educational Psychology Interactive*. Valdosta State University, Valdosta, GA, 2003.
7. Kuhn, D., Langer, J., Kohlberg, L., and Haan, N.S. (1977). The development of formal operations in logical and moral judgment. *Genetic Psychology Monographs 95* (1977), 97–188.
8. Magee, J. and Kramer, J. *Concurrency—State Models and Java Programs, 2nd Edition.* John Wiley & Sons, Chichester, UK, 2006.
9. Piaget, J. and Inhelder, B. *The Psychology of the Child*. Routledge & Kegan Paul, 1969.
10. *Webster's Third New International Dictionary*, 1966.
11. Wing, J.M. Computational thinking, *Comm of ACM 49*, 3 (Mar. 2006), 33–35.
12. Woolfolk and McCune-Nicolich. *Educational Psychology for Teachers, 2nd Edition*. Prentice-Hall, Englewood Cliffs, NJ, 1984.



**Jeff Kramer** (j.kramer@imperial.ac.uk) is dean of the Faculty of Engineering and a professor in the Department of Computing at Imperial College London.