Orit Hazzan's Column

Abstraction in Computer Science & Software Engineering: A Pedagogical Perspective

This column is coauthored with
Jeff Kramer, Department of Computing, Imperial College, London

ABSTRACT
Abstraction is recognized as a key concept in Computer Science and Software Engineering. Is it, however, possible to teach abstraction to students? This column discusses the role of abstraction in current curricula and the challenge posed in trying to teach it. We outline of kinds of tasks that may be used to enhance students' abstract thinking during their undergraduate studies.

1. INTRODUCTION
Abstraction has been discussed as a core concept of Computer Science (CS) and Software Engineering (SE) ever since the early days of the two disciplines. Its benefits and contributions to professionals in these fields have been acknowledged in different ways. For example, in many keynotes and conference presentations the noun *abstraction* and the verb *to abstract* are presented as useful heuristics that may be applied in a variety of situations and as a means to improve practitioners' professional skills in these fields. There are many explanations for the importance of the notion of abstraction. Among others, the intangibility of software systems, the need to abstract in order to deal with complexity, and the ability to examine many topics in CS and SE at different levels of detail and abstraction according to the purpose, are presented as strong justification for the central role of abstraction.

Although abstraction is a core concept of CS and SE, it is not an easy concept to teach. This fact can be explained in various ways.

First, abstraction is a soft conceptual idea. Unlike rigid concepts that are characterized by strict or formal rules, abstraction is similar to other concepts such as hierarchy, encapsulation and a programming style. Although abstraction can be formalized in specific circumstances (cf. refinement), abstraction in its broad sense does not lend itself to a full, comprehensive and concrete definition, nor are there any specific rules concerned with the application of abstraction. Furthermore, while examples should be given in order to illustrate the nature of abstraction, the danger is that when examples are actually given, the concept itself might be
obscured and might be conceived through the lens of the specific examples, thus losing its power which is largely expressed by its generality. On the other hand, if abstraction is presented using only its abstract definition, then students may find it difficult to grasp its meaning and may be unable to apply it in a meaningful manner.

Second, abstraction is not connected exclusively to a specific topic in CS or in SE. Rather, it is expressed throughout the curriculum and can, therefore, be illustrated and applied in the context of different topics. Consequently, educators face the question of whether to teach abstraction in a separate course dedicated to this topic, or to integrate the teaching of abstraction throughout the entire syllabus. This issue is addressed later in the column.

Third, since abstraction is never applied out of context, its application requires awareness. More specifically, abstraction is always utilized when another topic is at the focus of the discussion. In such situations it constitutes an accompanying, yet important, process that serves and supports the thinking concerning that other topic. Thus, when illustrating abstraction, one must, at the same time, teach that other topic, be aware of the application of abstraction and demonstrate the use of abstraction in problem-solving situations. Clearly, this is not a simple task.

Several questions naturally arise at this stage. How have those, who talk about abstraction and use it as a cognitive tool, actually grasped the concept? We believe that this understanding is a result of a long and reflective process that is part of the CS and SE professions. How, then, can we teach the concept of abstraction to students in the early stages of their studies, thus enabling them to conceive abstraction as a powerful technique and facilitate their ability to apply it meaningfully, from the early days of their professional lives? Clearly, such skills are bound to improve further through their professional experiences; however, if we improve students’ comprehension of the concept of abstraction from the early stages of their studies, we might improve their professional skills in general and their use of abstraction in particular. Finally, if we indeed agree that it is important to teach students the concept of abstraction as a core concept, the question is asked - How can we do it? This column attempts to provide the beginnings of an answer to this last question.

To this end, we suggest that abstraction should be introduced as an identifiable concept throughout the CS and SE curricula. Starting early in the CS and SE programs and continuing throughout, the concept of abstraction, its application and its usefulness in different courses and in different ways, should be highlighted. As an analogy to the concept of abstraction, consider the concept of proof as it is commonly taught in mathematics departments. Traditionally, although proofs are one of the very basic operations performed by mathematicians, the concept of proof itself, of argumentation and refutation, is often not taught explicitly at all. It is assumed that students will grasp its essence and additional related relevant knowledge, such as how to use proofs, what kinds of proofs exist, the limitations of proof and so on. In a similar way, such an assumption exists with respect to abstraction in CS and SE programs. This column suggests a different approach.
Section 2 of the column presents the concept of abstraction as it is presented in the CS and SE literature and in the educational literature of these fields. Section 3 presents patterns of questions about abstraction that can be integrated throughout the CS and SE curricula.

2. ABSTRACTION IN CS AND SE

2.1. The Concept of Abstraction in CS and SE

Abstraction is a central topic both in CS and in SE. It is a cognitive means according to which, in order to overcome complexity at a specific stage of a problem solving situation, we concentrate on the essential features of our subject of thought, and ignore irrelevant details. Abstraction is especially essential in solving complex problems as it enables the problem solver to think in terms of conceptual ideas rather than in terms of their details. Abstraction can be expressed in different ways. Liskov and Guttag (1986), for example, describe abstraction in program development as “a way to do decomposition productively by changing the level of detail to be considered. ... The process of abstraction can be seen as an application of many-to-one mapping. It allows us to forget information and consequently to treat things that are different as if they were the same. We do this in the hope of simplifying our analysis by separating attributes that are relevant from those that are not”. (p. 3). The decision as to what is essential and what is irrelevant depends on the purpose of the abstraction. For a full examination of different kinds of abstraction see Liskov and Guttag (2001).

In addition to the use of abstraction, CS and SE practitioners should be aware of the need to think in terms of different abstraction levels and capable of moving between abstraction levels. For example, software developers move from a global view of the system (high level of abstraction) to a local, detailed view of the system (low level of abstraction), and vice versa. When trying to understand customers’ requirements, developers have a global view of the application (high level of abstraction); whereas when coding a specific class, a local perspective (on a lower abstraction level) is adopted. Obviously, there are many intermediate abstraction levels in between these two levels, which programmers consider during the process of software development. The knowledge of how and when to move between different levels of abstraction does not, however, always come naturally, and requires some degree of awareness and experience. For example, a software developer may remain in too low a level of abstraction for too long, while the problem he or she faces could have been solved more readily had it been viewed at a more appropriate level of abstraction. The shift to that abstraction level might not be a natural move unless one is aware that this might offer a step towards a solution.

2.2. The Concept of Abstraction in CS and SE Education

This section examines how abstraction is addressed in the Computing Curricula 2001 documents.

The second principle guiding the Computing Curricula 2001 Task Force is

2. Computer science draws its foundations from a wide variety of disciplines.

Undergraduate study of computer science requires students to utilize concepts from many different fields. All computer science students must learn to integrate theory and practice, to
recognize the importance of abstraction, and to appreciate the value of good engineering design. (p. 12)

In practice, it is recommended that the core topic of abstraction mechanisms (PL5) is presented in a minimum core coverage time of three hours. An examination of the suggested teaching topics reveals that abstraction is addressed through its applications, but not with respect to its essence. For example, parameterization mechanisms (reference vs. value) is one such topic. One of the Learning Objectives associated with it is: "Defend the importance of abstractions, especially with respect to programming-in-the-large." (p. 115)

In the Software Engineering 2004 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, which is a volume of the Computing Curricula Series published in August 23, 2004, it is explained that:

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 of this challenge is that the supporting process must be applied at multiple levels of abstraction. (p. 8)

Similar to the approach adopted in the CS volume, abstraction is presented as one of the core principles guiding the development of the SE Guidelines volume:

[2] Software Engineering draws its foundations from a wide variety of disciplines.... All software engineering students must learn to integrate theory and practice, to recognize the importance of abstraction and modeling, to be able to acquire special domain knowledge beyond the computing discipline for the purposes of supporting software development in specific domains of application, and to appreciate the value of good design. (p. 19)

Furthermore, Curriculum Guideline 4 states that:

Many SE concepts, principles, and issues should be taught as recurring themes throughout the curriculum to help students develop a software engineering mindset.... In addition to ethics and tool use, which will be highlighted specifically in other guidelines, the following are types of material that should be presented, at least in part, as recurring themes:

- Measurement, quantification, and formal or mathematical approaches.
- Modeling, representation, and abstraction.
- Human factors and usability: Students need to repeatedly see how software engineering is not just about technology. (p. 38)

However, the actual teaching of abstraction suggested by the Software Engineering 2004 Curriculum Guidelines is limited to several hours as follows: In the Object-Oriented Paradigm course - 3 core hours; in the Data Structures and Algorithms course - 2 core hours; in the Introduction to Software Engineering and Computing course - 1 core hour; in the Software Engineering and Computing II course - 1 core hour.
The above brief discussion on the nature of abstraction indicates that abstraction is a central concept in CS and in SE and that it should be highlighted throughout the curriculum of these fields. Such an observation leads to the question: How then can we give abstraction the attention it deserves? Is it possible to highlight abstraction throughout these curricula? A possible approach is suggested next.

3. QUESTION PATTERNS FOR ABSTRACTION

This section presents a collection of kinds of questions that might serve as a means to teach abstraction throughout the curriculum. Since the questions are presented as question patterns, instructors would be required to instantiate them in the context of the specific course that they teach, i.e. to elaborate the question details according to the subject of the course. With each kind of question, an illustrative example is presented, and the way in which it might enhance students’ abstraction skills is explained.

Pattern 1: Given two representations of a specific system, students are asked to explain which representation is more abstract and why. These representations could be program code, a UML diagram, a written description, a photo and so on.

Example: The following two representations describe the same system. In your opinion, which representation is more abstract? Explain your opinion.

| A verbal description of a system that reflects a high level of abstraction | A detailed drawing of the same system that reflects a lower level of abstraction |

Explanation: This kind of task requests students to analyze what is abstract and why. In doing so, they must address the essence of abstraction and the features of an abstract object.

Pattern 2: As in Pattern 1, but with more than two representations. The students’ task is to rank the given representations according to their level of abstraction.

Example: To the above two representations add two UML diagrams: A class diagram and a sequence diagram.

Explanation: As well as examining these representations as described in the previous pattern of questions, the addition of the two diagrams encourages students to rethink and compare static and dynamic representations of a given system through the lens of abstraction. It also raises the need to consider purpose as part of abstraction.

Pattern 3: Categorization of a collection of representations of different systems according to the student’s choice of abstraction.

Example: Following are eight representations of 3 systems. Group these representations in at least two ways and explain the rationale for your groupings.

Explanation: As in the previous questions, these representations can be of different kinds. In this case, in addition to examining the given representations as abstractions of the given systems, students must also analyze similarities and differences across systems, and hence apply a different abstraction mechanism. By doing so, students not only abstract, but also refer to different ways in which abstractions can be expressed.
**Pattern 4:** Description of a specific system, with which the students are familiar, at different levels of abstraction.

**Example:** Describe your cell phone on at least four levels of abstraction. Outline the main considerations that guided you in choosing these descriptions.

**Explanation:** In these tasks students are asked to present a single system in terms of several different levels of abstraction. In order to do so, they must clarify, in that specific context, the meaning and purpose of different abstraction levels and how these levels of abstraction are reflected in their representations.

**Pattern 5:** Given a system representation, students are asked to give one representation that is more abstract than the one given and one representation that is less abstract than the one given.

**Example:** The following representation describes a GPS system. Describe the system in two additional ways: one that describes it in terms of a higher level of abstraction and one that describes it in terms of a lower level of abstraction.

**Explanation:** Similar to the previous question the fact that students are asked to rank representations according to their abstraction levels prompts them to examine these representations through the lens of abstraction and to examine what makes one representation more or less abstract than another. If instructors wish to refer not only to the level of details when examining abstraction, students could also be asked to present descriptions of a different kind than the one given. For example, if the given representation is verbal, students may be asked to add visual representations.

**Pattern 6:** Students are asked to explain some topic X (a system, a machine, and so on) to someone else. They are asked to describe how they would explain X in two cases: when the person they are explaining it to is able to see X and when he or she cannot see X. They are also asked to explain the considerations that guided the formulation of each description and how these formulations relate to different levels of abstraction.

**Example:** Choose an IDE with which you are familiar but your friend is not. Describe this environment to your friend in the two following cases: in the first situation assume that you are sitting with your friend in front of a computer screen; in the second case your task is to explain the IDE to your friend over the phone. Reflect: What elements would you emphasize in each case? Why? What role does abstraction play and how is it expressed in each description?

**Explanation:** This question invites students to examine, from an abstract perspective, how their descriptions are influenced by their interaction level with the other person. Such an examination is important because abstraction, as a means for reducing complexity, also plays a significant role in human communication.

**Pattern 7:** Students are asked to explain some topic X (a system, a machine, and so on) with a certain constraint or constraints. Students are also asked to explain their considerations in choosing their explanations.
Example: Choose an IDE with which you are familiar but your friend is not. Describe this environment to your friend without using the mouse.

Explanation: This question invites students to examine, through the lens of abstraction, how their ways of communicating are influenced by the tools available to them. Similar tasks can be constructed in different contexts, such as: develop a computer program without using procedures; design a system without using inheritance, and so on.

Pattern 8: Students are asked to suggest a metaphor or an analogy to a given system.

Example: Find a metaphor for the software system you are developing now.

Explanation: Since metaphors capture similarities (to some degree) between two (or more) systems, they naturally can serve as a way to elicit abstraction processes.

Pattern 9: Given a problem, ask students to describe two processes for its solution: one that is guided by abstraction and another that makes no use whatsoever of abstraction. Ask the students to compare the two processes and to draw conclusions.

Example: Imagine you are asked to develop a system that does X. Describe two processes by which this system can be developed. The first process is constantly guided by abstraction; the second uses no abstraction. After completing the formulation of the two processes, compare them and draw your conclusions.

Explanation: Naturally, the more complex the system is, the more crucial is the use of abstraction. Thus, students cannot ignore it. Furthermore, the reflection and the comparison that follow the creation process also enhance students’ awareness to the process of abstraction.

Pattern 10: Reflection questions: this kind of questions can be used in many situations. They invite students to reflect on their thinking processes and by doing so increase the abstraction level of their mental processes. This reflective practice perspective was introduced by Schön (1983, 1987) and was applied to SE by Hazzan (2002) and by Hazzan and Tomayko (2005).

Example: Reflect on the last software project you developed. If you had to repeat the task, would you do it in the same way? Why would you change what you decided to change?

Explanation: By asking students to rethink their creative processes, students must increase the abstraction level of the objects they consider. Specifically, instead of thinking in terms of the objects students work with, mental processes and considerations become the central objects of thoughts.

A general note about the patterns: The above patterns of questions may address different topics through the lens of abstraction. Instructors can complete the details according to the course contents. It is important to mention that such open questions require careful use so as to avoid a situation in which students give standard answers. We would not want our students, for example, to simply repeat considerations for using abstraction that they have heard in class.
4. CONCLUSION

This column suggests patterns of questions that would require students to perform abstract-oriented analysis. Details of the questions would have to be completed according to the contents of specific courses. Also, they will undoubtedly need further refinement and evaluation.

As hinted previously, abstraction is in itself an abstract concept (pun intended). Therefore, as with other abstract concepts, simply lecturing on the topic is unlikely to be comprehensible to students. In order to perceive concepts such as abstraction, students must first see, comprehend and experience the benefit from its use, and only then can they discuss and abstract the concepts' qualifications. In the following, we suggest two processes by which abstraction can be presented to students.

The first is based on the assumption that CS and SE educators apply abstraction (sometimes maybe even unconsciously). Thus, instructors can simply illustrate the actual process of abstracting to their students. Indeed, such an illustration requires awareness of one's own mental processes. In practice, while performing this in front of the class, instructors can specifically make statements such as, "I ignore this aspect here because…", "Now, let's go one level of abstraction down and elaborate …", "In a similar way to what we did last week, we can…", "Abstraction is expressed here because…", "If we hadn't used abstraction the solution would …", and so on. This can be done, for example, when instructors develop a solution in stages in front of the class; instead of presenting the students with complete solutions, they can specify how abstraction is expressed and how it guides the solution's progress.

The second teaching process integrates reflective processes. Such reflection should be conducted together with the students, and may address, among other issues, topics such as the way abstraction is expressed in different situations, and questions such as: What would have happened had we not used abstraction? Such demonstrations might also increase students' awareness with respect to students' reflective skills.

Another pedagogical issue that should be examined is whether to dedicate a full course to abstraction (maybe together with other heuristics) or rather to mention abstraction on different occasions in different courses, whenever appropriate. Although the argument presented in this column indicates a preference for the latter option, a dedicated abstraction course may be appropriate in some circumstances. It would, of course, highlight the importance of abstraction and enable students to concentrate on the topic without the time pressure that characterizes many courses. On the other hand, such conceptual (or philosophical) courses can be very difficult to teach and difficult for students to comprehend without a high degree of maturity in the topic. Further, such a course may even alienate students, giving them the impression that abstraction is not a practical problem solving technique. As in many cases, a good solution would integrate the two options. Specifically, instructors should certainly address abstraction in all courses, when appropriate; and in parallel, an elective course can be offered for those students who find it an interesting topic. Since fewer students usually enroll in elective courses, instructors might be able to facilitate discussions and reflection processes in more supportive manner.
It is interesting to consider whether or not the proposals presented in this column will indeed enhance students' abstraction skills. Personal experience, which must of course be subjected to further scrutiny, suggests that such an abstraction-oriented education should increase students' awareness of abstraction and improve their ability to apply it. For example, during a Programming Paradigms course in which the approach described in this column is applied, students are frequently heard making statements such as: "We guess we shouldn't move to lower level of abstraction now", or "It is analyzed from an abstract perspective". It might be added that the tension between code-level instruction and abstract instruction never completely disappears; In fact, on occasion students still tend to reduce abstraction (Hazzan, 2003), focusing on low-level specific details, and are not ready to capture the power and usefulness of abstraction. Such occasions can be used, however, to further highlight the tension between different levels and purposes of abstraction, addressing its source and ways in which it can be bridged.

A sequel to the present column might deal with the issue of testing abstraction skills. Kramer (in press) proposes that "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". Accordingly, Kramer suggests that we need tests to assess those abilities and skills and presents several usages for testing abstraction. One of them is the annual measurement of students' abstraction abilities while at college so as to gain insight into the effectiveness of our teaching techniques; another is the measurement of students' abstraction skills when applying to study CS or SE so as to assess their aptitude. Perhaps such tests could be based on the question patterns described in this column, including applications in other contexts, such as mathematics, architecture or art.

5. REFERENCES


Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science, Computer Science Education 13(2), pp. 95-122.


