Teaching Abstraction in Introductory Courses

Herman Koppelman, Department of Computer Science, University of Twente, Enschede, the Netherlands, Department of Computer Science, Open University the Netherlands

Betsy van Dijk, Department of Computer Science, University of Twente, Enschede, the Netherlands

15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'10). June 28-30, 2010, Ankara, Turkey

ABSTRACT

Abstraction is viewed as a key concept in computer science. It is not only an important concept but also one that is difficult to master. This paper focuses on the problems that novices experience when they first encounter this concept. Three assignments from introductory courses are analyzed, to understand why abstraction is difficult for novices. This analysis leads to a number of guidelines that can be used by instructors to support novices learning abstraction.

Keywords

Abstraction, abstraction level, computer science, pedagogy, recursion.

1. INTRODUCTION

Abstraction is seen as a key concept in computer science [1, 5, 11]. Already in the early days of the discipline, E.W. Dijkstra wrote: 'We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases is called "abstraction"; 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.' [4]

Viewing abstraction as a key concept is confirmed by the ACM/IEEE report Computer Science Curriculum 2008: An Interim Revision of CS 2001 [2]. The report stresses that abstraction should be highlighted throughout the curriculum. One of the principles that guided the work of the Task Force that wrote the report is that all computer science students must learn to recognize the importance of abstraction. According to the report one of the expected characteristics of computer science graduates is familiarity with common themes and principles. Abstraction is mentioned as one of the recurring themes that students should encounter in the course of an undergraduate program in computer science.

Abstraction can have many 'faces', even within the context of computer science. An interesting and extensive discussion about the faces of abstraction in mathematics and computer science can be found in [5]. An important aspect of abstraction for computer science is 'ignoring the details', as can be concluded from the following descriptions and characterizations of abstraction:

- abstraction concentrates on the essential features of a subject of thought and ignores irrelevant details [10]
- abstraction is the act of withdrawing or removing something
- [11]
- abstraction is the act or process of leaving out of consideration one or more properties of a complex object so as to attend to others [11]
- you use abstraction when you do know a lot and you want to suppress details to get a better idea of what's going on [5].

Abstraction plays a role in many sub disciplines of computer science, such as human-computer interaction, requirements engineering, databases and software design. In this paper we focus on the use of abstraction within the domain of software design.

In the context of software design, abstraction makes it possible for the software engineer to think in terms of concepts rather than in terms of their implementation details. This is useful in solving complex problems, as it allows one to concentrate on essential features of an object and to ignore irrelevant details. Details can be hidden, enabling the problem solver to get a better idea of what is going on. Therefore abstraction skills are essential in the construction of appropriate models, designs and implementations [10]. For example, programmers can use concepts such as methods, procedures, functions, classes and objects without worrying about how they will do their jobs. In this way the programmer is relieved from the burden of having to be aware of the entire program at several levels at all times.

Electronic copy available at: https://ssrn.com/abstract=2953482

In the context of software design, not only the concept of abstraction is important but also the concept of abstraction *level*. Software engineers should be able to think in terms of different abstraction levels and to move between them. The capabilities to rapidly change the level of abstraction, to concentrate on one level at a time, to see things subsequently 'in the large' and 'in the small' [9] are seen as characteristics of computer scientists. In [9] it is argued that in computer science the scale changes through many orders of magnitude, from the individual bits to the complex constructs of languages.

It is not possible to be a skilled software engineer without having at one's disposal the skill of abstraction. In [10] it is stated that once you realize that computing is all about abstractions, it becomes clear that an important prerequisite for writing good computer programs is the ability to handle abstractions in a precise manner.

Studies show that experts and novices in the field of computer science differ in their abstraction skills [8]. In [11] it is even suggested that the ability to use abstraction skills explains the difference between software engineers who produce clear and elegant designs and programs, and those who do not.

Abstraction is not only a key concept from the professional point of view, but also from the learner's point of view. In an empirical study [12] it was found that students often describe situations in which they applied different kinds of abstraction as transformative experiments, that is, as experiments that transform the way they see and experience computing.

Abstraction is a key concept in computer science, but also a complex concept, which is not easy to learn [10]. There are even doubts whether abstraction is something that can be learned at all. 'Is abstraction teachable at all? Isn't it something one has or doesn't have? (...) some will get it, many will not, and a few will be very good at it.' [10] 'Is it possible to improve abstract thinking and abstraction skills by education?' [11]

It can be concluded that teaching abstraction is a pedagogical challenge. In this paper we focus upon the challenge of teaching abstraction to novices. The main questions in this paper are:

- which problems do novices encounter when they learn abstraction?
- in which way can instructors support novices learning abstraction?

We discuss these questions by analysing, in section 2, three simple programming problems for which it is beneficial to use abstraction to solve them. We analyse which forces make it hard for novices to actually use abstraction. In section 3 we discuss some pedagogical consequences of our analysis.

2. ANALYSIS OF THREE PROBLEMS

2.1 Problem 1: a Nested Loop

The first is an introductory programming problem. Given is an array a[1..100] of integers. The assignment is to find out if a has an element a[i] with the property: the sum of the digits of a[i] equals 10.

A possible algorithm is:

```
i := 1;
found := FALSE;
WHILE (i < 101) AND NOT found DO
  number := a[i];
  sum := 0;
  WHILE number > 9 AND sum < 10 DO
    lastDigit := number MOD 10;
    number := number DIV 10;
    sum := sum + lastDigit;
    found := sum = 10;
  END;
  i := i + 1;
END;
```

A crucial feature of this algorithm is the nested while-loop it contains. Nested while-loops generate a lot of mistakes. For example [13] presents an interesting overview and analysis of problems students have with a nested loop that is similar to the one discussed here. In an experiment novices did not usually recognize the need for a nested loop. And if they did, they had problems separating the conditions that control both loops.

A major problem with the presented algorithm can be analyzed in terms of abstraction and abstraction levels. The algorithm has several abstraction levels, but two of them are relevant in this context. The first level pertains to a given array of numbers and the process of navigating systematically through this array in search of an element with a property that is not yet specified. At the second level this property is specified and a procedure is described to determine whether a given integer has the property.

In the algorithm these levels of abstraction are mixed. Reading the algorithm from start to finish, one jumps to and fro between those levels. Starting at the level of the array (i := 1), one goes to the second level from the statement 'sum := 0', to finish at the first level (i := i + 1).

The relevant levels can be neatly separated by introducing a function as an abstraction mechanism:

```
i := 1;
found := FALSE;
WHILE (i < 101) AND NOT found DO
  found := Satisfies (a[i]);
  i := i + 1;
END;
```

The algorithm is the same but now an abstraction mechanism has been used to separate the abstraction levels. As a consequence the programmer can concentrate on one level at a time. The abstraction is that the programmer does not have to care about the way the function Satisfies determines whether an integer has the required property. In this way students do not have to deal simultaneously with different levels of abstraction.

Electronic copy available at: https://ssrn.com/abstract=2953482

Novices do not usually spontaneously come up with this function. For example in [13], which reports about an empirical study of the problems students have with a similar nested loop, it has not been found. That is not surprising. Let us compare how humans would tackle this problem. A systematic description of the way humans would solve this problem by hand, is:

- look at the first element of the array, sum the digits and check if this sum is 10
- if so stop
  otherwise continue in the same way with the next elements.

In this solution-by-hand the relevant abstraction levels are mixed. Coding it into a programming language leads to the nested loop solution as a matter of course.

Therefore, there is a good reason why the use of abstraction in this case is not natural for novices. It requires thinking about the problem in a new and different way. As a consequence, the use of abstraction in this example has to be taught explicitly.

Teaching this problem offers a good opportunity to introduce or illustrate the concept of abstraction. Students can be shown that it is beneficial to use abstraction in this example, from a practical point of view. Hiding the nested loop makes the solution clearer and making mistakes less probable. David Gries [7] even advocates *'always* to hide nested loops' in this fashion. (Italics by Gries.) Unfortunately, in textbooks it is not common practice to do so. Usually one finds nested loops in cases such as these, without any reference to the concept of abstraction.

2.2 Problem 2: a Nested SQL Query

The second problem stems from the domain of an introductory database course. Two tables of a relational database are given, one about flights and the other about airplanes. An example is shown in figure 1.

Table Flight

flight nr	*departure*	*destination*	*plane nr*
265	Rome	Athens	13
415	Berlin	Paris	28
598	Oslo	Sofia	17

Table Plane

plane nr	*capacity*	*company*	*type*
13	235	Air Berlin	Airbus 310
17	210	Iberia	Airbus 310
28	240	Alitalia	Boeing 747

Figure 1: Tables Flight and Plane

The assignment is to write an SQL-query to find the flight numbers and the places of departure and destination of flights performed by an Airbus 310.

A straight ahead solution with a nested query is:

```
select flight_nr, departure, destination
from   Flight
where  plane_nr in (select plane_nr
                    from   Plane
                    where  type='Airbus310')
```

In this nested query two levels of abstraction are essential. The first level pertains to the main SQL-query. It lists the required attributes and the tables in which they can be found. This level abstracts from the way to find the relevant plane numbers and considers the way to find these numbers as details, to be found out next. Informally this highest level can be written as

```
select flight_nr, departure, destination
from   Flight
where  'planes have type Airbus 310'
```

The second level pertains to the sub query, which describes finding the planes of the required type.

Again we compare this solution with the way humans are inclined to solve this problem by hand. A systematic solution is:

- highlight the rows of table Plane with type Airbus 310
- extract the set of plane numbers of the highlighted rows
- highlight all the rows in table Flight that have a plane number in the resulting set
- extract the required attributes from the highlighted rows of table Flight.

This solution strongly suggests starting at the second level and only going to the first level afterwards. There is no straightforward translation of this solution to the nested SQL- query where the levels are neatly separated and where one starts at the highest level.

SQL is a declarative language and the programmer is forced to focus upon the 'what'. Of course, for the novice programmer this problem is not very complicated. But the way of describing the solution is new and unfamiliar. This gives instructors the opportunity to illustrate explicitly the concept of abstraction in the context of writing SQL queries and to discuss the differences between the SQL-query and the 'solution by hand'.

2.3 Problem 3: a Recursive Function

Recursion is considered as a difficult concept to learn. Many students have a notoriously difficult time learning to program recursion. Eric Roberts observes that for the student recursion appears to be obscure, difficult, and mystical [14]. For that reason it is a well-known topic within the computer science education literature. The problems students experience in understanding recursion have been studied extensively, for example in [6, 16].

As a simple example we discuss the function NumberOfDigits, which counts the number of digits of a given positive integer. A recursive implementation is:

```
NumberOfDigits (posint n): posint
IF n < 10 THEN
   RETURN 1
ELSE
   RETURN NumberOfDigits (n div 10) + 1
```

Novices have problems verifying such an implementation. They have a preference for simulating the process of the successive calls of the recursive function [6, 16]. They tend to take an example, such as NumberOfDigits(23478), and to write out all of the recursive function calls. They check the flow of control and do by hand what the processor does. A function call is considered to be a way to move the control flow to another part of the program.

The key to comprehending recursion is to be able to abstract from the implementation of the recursive function call. One has to stop on seeing the function call and consider the effect of the function call, not the way the effect is realized. One should be able to think about *what* needs to be done and temporarily suspend thinking about *how* it is done. The abstraction is that one must learn to stop analyzing the process after the first function call. The rest of the problem will take care of itself [14].

For many novices this is a new way of looking at functions. Eric Roberts [14] speaks about the 'recursive leap of faith'. This 'faith' is not acquired spontaneously but has to be learnt.

Unfortunately, textbooks do not usually support novices in mastering this perspective. On the contrary, they do not emphasize the use of abstraction, but they support the students' inclination to write out all the successive recursive function calls. Examples are [3, 15], but many more books have this approach.

The use of the flow-of-control approach is limited. One cannot demonstrate the correctness of a program by checking an example. This approach is not wrong, of course, but it should be complemented by teaching explicitly the 'recursive leap of faith': to trust that a recursive function call does what it promises.

It is not only *verifying* a recursive algorithm that poses problems to novices, *designing* it does as well. To find a recursive algorithm one has to use abstraction. In the given example one has to see an integer as a digit followed by an integer or as an integer followed by a digit. For example, the number 290111952 has to be seen as:

29011952 = 2 9011952, or

29011952 = 2901195 2

One has to abstract from the digits and see a series of digits as an integer. From many studies we know that novices have problems with this perspective. Usually novices see an iterative solution first [16], and for a good reason. If one counts the number of digits by hand, one sees the integer as a sequence of digits, and counts them one by one.

29011952 = 2 9 0 1 1 9 5 2

This perspective hampers finding a recursive solution. One has to abstract from this view, and see an integer not as a series of digits, but as an integer followed by a digit. Therefore, in this example the use of abstraction to find a solution does not come naturally. One has to learn it.

Recursion is an unfamiliar idea and requires thinking about problems in a new and different way. A recursive solution in no way resembles the way humans would solve a problem by hand. To really understand recursion and be able to apply it, one has to master the concept of abstraction. Therefore, recursion is preeminently a means that can help hone students' abstraction skills. The key is not to think too hard and not to care about implementation details. In [14] it is explained that most people's minds rebel a bit at trusting that a recursive function call does what it promises.

Unfortunately, many textbooks (examples are again [3, 15]) focus on descriptions of the processes generated by recursive function calls, and not on the abstraction aspect.

3. DISCUSSION

The examples we have discussed suggest that the use of abstraction in programs is not straightforward for novices. Modern languages offer a lot of abstraction mechanisms. Programmers are invited to use them. But it is not enough just to offer abstraction mechanisms. More is needed to have novices develop abstraction skills, because forces exist that prevent novices from using abstraction spontaneously. If they have to solve a new problem, novices tend to rely on a familiar procedure. A familiar strategy to solve a problem is to realise how to solve it by hand, systemize it and code it. In many textbooks this strategy is promoted, sometimes explicitly. For example in [15] novices are stimulated to describe algorithms in natural language, 'as if the instructions were to be given to a human being'. Textbook [3] gives as a hint for finding an algorithm: 'How would you solve the problem by hand?'

The examples we analyzed showed that there is a gap between an algorithm based upon 'solving by hand' and the use of abstraction. In the case of recursion the algorithm is even completely different. More generally there is a gap between the way introductory programming is taught and mastering the skills of abstraction. Introductory programming courses promote the understanding of the flow of control of a program. Flow charts are used to explain the flow of control, flow of control is simulated 'by hand' or by visualization tools, backtracking procedures are offered to understand recursive function calls, the use of role playing is promoted as a didactical technique to depict the actions of an algorithm.

There is nothing wrong with this 'flow of control' view. It promotes understanding of the way algorithms are executed. There is no doubt that Computer Science students should thoroughly understand this. But students should also learn to see all kinds of abstractions in a piece of code. They should be taught to see a procedure or function call not only as a way to transfer control, but also as a mechanism for suppressing irrelevant detail. They should be able to focus upon the effect of a procedure or function call and to resist the inclination to check the implementation. As we analysed, especially in the case of recursive procedures it takes effort to conquer this perspective.

What does our analysis suggest for the pedagogy of teaching abstraction to novices? First of all we recommend that teaching abstraction should start early. The discussed examples show that the concept of abstraction can be introduced and explained with simple problems. Abstraction is not a concept one masters or not. In the empirical study [12] it was found that many students learn gradually about concepts such as abstraction through concrete examples. Therefore it takes time before students fully understand the concept.

The next recommendation is to teach abstraction consciously. Expert computer scientists use all kinds of abstractions, usually

without realizing it. Teaching consciously means that instructors should point out where abstraction has been used and call it by its name. Instructors should also show why a particular abstraction in a concrete example is new for the novices. Sometimes students have to get rid of some engrained habits.

More generally, teaching programming is not only about features of a language, but also about principles and strategies that should be applied upon program design. Gries observes: 'Programming is more than a bunch of facts. It is a skill (...) teaching programming as a skill means more than teaching facts (...).' [7]. Gries also observes that it is not common practice that textbooks support this approach, and that many modem textbooks teach just programming, not the concepts involved [7].

The third recommendation is to stress the benefits of using abstraction. For many students abstraction has the connotation of being hard and obscure. In empirical study [5] students characterized abstraction as being complex, requiring deep thought, not easy to understand. To cure this idea, instructors can show that abstraction makes life easier, once one has mastered that concept. Instructors can compare solutions with and without a specific application of abstraction. They can show the benefits of using abstraction. For example, algorithms are easier to write and understand. Recursive algorithms, especially, can be surprisingly elegant and simple.

4. CONCLUSION

Abstraction is a key computer science concept. It deserves to play a central role in the computer science curriculum. Our analysis suggests that within the context of software design abstraction is not a concept that is easy to master. It is new and difficult for novices, and conflicts with a natural problem solving strategy: analyse how to solve a problem 'by hand' and implement that. Using abstraction means the need to look at problems in a new way. Therefore, novices will not get a deep understanding of abstraction spontaneously; they have to take pains to learn the concept.

We recommend teaching the concept of abstraction from the beginning. We showed three examples of problems that give the opportunity to illustrate the concept of abstraction in introductory courses. We also recommend teaching abstraction explicitly. Instructors should point out where and how abstraction is used. For example, they can show where and why separating the levels of abstraction works better. Finally, we recommend stressing the advantages of using abstraction. Its usefulness in different ways should be highlighted. The idea that abstraction is complex and hard to understand has to be removed.

5. REFERENCES

[1] Bennedsen, J., and Caspersen, M. E. 2008. Abstraction ability as an indicator of success for learning computing science? In *Proceeding of the Fourth international Workshop on Computing Education Research* (Sydney, Australia, September 06 - 07, 2008). ICER '08. ACM, New York, NY, 15-26. DOI= http://doi.acm.org/10.1145/1404520.1404523 .

[2] Computer Science Curriculum 2008: An Interim Revision of CS 2001, Report from the Interim Review Task Force, http://www.acm.org//education/curricula/ComputerScience2008.pdf

[3] Dale, N.B., and Lewis, J. 2006. Computer Science Illuminated, Jones and Bartlett Publishers, Inc.

[4] Dijkstra, E. W. 2007. The humble programmer. In *ACM Turing Award Lectures* ACM, New York, NY, 1972. DOI= http://doi.acm.org/10.1145/1283920.1283927.

[5] Frorer, P., Hazzan, O., and Manes, M. 1997. Revealing the faces of abstraction. *International Journal of Computers for Mathematical Learning*, 2(3), 217-228.

[6] Ginat, D. 2004. Do senior CS students capitalize on recursion? In *Proceedings of the 9th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (Leeds, United Kingdom, June 28 - 30, 2004). ITiCSE '04. ACM, New York, NY, 82-86. DOI= http://doi.acm.org/10.1145/1007996.1008020

[7] Gries, D. 2002. Where is programming methodology these days? *SIGCSE Bull.* 34, 4 (Dec. 2002), 5-7. DOI= http://doi.acm.org/10.1145/820127.820129

[8] Haberman, B. 2004. High-School Students' Attitudes Regarding Procedural Abstraction. *Education and Information Technologies* 9, 2 (Jun. 2004), 131-145. DOI= http://dx.doi.org/10.1023/B:EAIT.0000027926.99053.6f

[9] Hartmanis, J. 1994. Turing Award lecture on computational complexity and the nature of computer science. *Commun. ACM* 37, 10 (Oct. 1994), 37-43. DOI= http://doi.acm.org/10.1145/194313.214781

[10] Kramer, J. and Hazzan, O. 2006. The role of abstraction in software engineering. In *Proceedings of the 28th international Conference on Software Engineering* (Shanghai, China, May 20 - 28, 2006). ICSE '06. ACM, New York, NY, 1017-1018. DOI= http://doi.acm.org/10.1145/1134285.1134481

[11] Kramer, J. 2007. Is abstraction the key to computing? *Commun. ACM* 50, 4 (Apr. 2007), 36-42. DOI= http://doi.acm.org/10.1145/1232743.1232745 .

[12] Mostrom, J. E., Boustedt, J., Eckerdal, A., McCartney, R., Sanders, K., Thomas, L., and Zander, C. 2008. Concrete examples of abstraction as manifested in students' transformative experiences. In *Proceeding of 4th international Workshop on Computing Education Research* (Sydney, Australia, Sept 06-07, 2008). ICER '08. ACM, New York, NY, 125-136. DOI= http://doi.acm.org/10.1145/1404520.140453 .

[13] Muller, O., Ginat, D., and Haberman, B. 2007. Pattern-oriented instruction and its influence on problem decomposition and solution construction. *SIGCSE Bull.* 39, 3 (Jun. 2007), 151-155. DOI= http://doi.acm.org/10.1145/1269900.1268830 .

[14] Roberts, E.S. 2005 Thinking Recursively: with Examples in Java. John Wiley & Sons.

[15] Savitch, W. 2008 Problem Solving with C++ (7th Edition), Addison-Wesley.

[16] Sooriamurthi, R. 2001. Problems in comprehending recursion and suggested solutions. In *Proceedings of the 6th Annual Conference on innovation and Technology in Computer Science Education* (Canterbury, United Kingdom). ITiCSE '01. ACM, New York, NY, 25-28. DOI= http://doi.acm.org/10.1145/377435.377458