**Understanding ‘Abstraction’ and its Importance in Teaching Computer Science to the First-Learners**

**Abstract:** Abstraction is a process of taking away or hiding some detail features of something in order to give more attention to its some other essential features. It is a fundamental problem solving skill in mathematics and computer science as well as general learning. There is no argument about the importance of abstraction; however, there is no unique consensus of how abstraction can be taught in different perspectives. Moreover, teaching abstraction is not an easy task and there is no universally accepted technique or tool that has been derived yet to teach this essential skill to computer science first-learners, especially high school students and prospective computer science majors. In this paper, we define and describe the underlying concepts of abstraction across several disciplines, and then narrow the focus to the context of computer science. The discussion is driven by the importance of abstraction in the high school and early undergraduate curricula. The paper contributes by offering suggestions for teaching abstraction skills to prospective computer science students through the use of simple mathematical problems.

**Keywords**

Abstraction, AP Computer Science Priciples, Computer Science Education, First-Learners

**Introduction**

Abstraction is the purposeful suppression, or hiding of some details of a process or an object with an intention to paying more attention to some other aspects, details, or structure (Budd, 2002). The word “abstraction” is derived from the word “abstract” which is the opposite of “concrete.” Abstract things are sometimes defined as those things that do not exist in reality. Concrete thinking refers to thinking on the surface whereas abstract thinking is related to thinking in depth. Concrete thinking often lacks depth and refers to thinking in the periphery. On the other hand, abstract thinking is based on thinking about an idea that goes under the surface. It is all about understanding multiple meanings whereas concrete thinking is based on using one’s senses. For instance, by looking at the Statue of Liberty, someone applying concrete thinking may consider it as only a statue of a lady holding a torch. However, a person exercising abstract thinking will consider the statue as a symbol of freedom, liberty, and hope. Over time and circumstances, the meanings of abstract terms may change, but the nature of concrete things stay more constant. For instance, the word “love” may imply different connotations based on one’s age. The abstract concept of love changes as we mature and go through different life experiences. However, concrete terms (e.g., “hot,” “spoon”) may suggest the same meanings to us regardless of our age.

In education, abstraction is a process by which learners develop concepts either from previous experience or from other concepts they already know (Kramer, 2007). It is an inductive process of forming an abstract concept by identifying common features among a group of concrete objects, or by ignoring unique aspects of those concrete objects. In cognitive psychology, abstraction is a process of extracting the commonalities from a set of objects and then naming these defining features. For instance, if we consider a chair and a table as two concrete objects, one of their common features is that they are both members of a superordinate set called “furniture.” Furniture is the “superordinate” abstraction of those two “subordinate” objects and may then include other objects not previously listed, like the arms and legs of the chair, or the top of the table (Rosch & Mervis, 1975).

In mathematics, abstraction is a fundamental process where development of inference structures through information negligence is primarily focus on (Colburn & Shute, 2007). It is a process of extracting the underlying essence of a mathematical concept by removing its dependencies on other real-world objects with which it might originally be connected; and generalizing it to apply to other abstract descriptions of equivalent phenomena. Mathematics uses many everyday words in an abstract form such that their meaning is defined precisely in relation to other mathematical terms- but not by their everyday meaning. For example, although everyday language occasionally uses symbols like *x* and *y*, objects like *xy* and √(-1) are unknown or rarely used outside mathematics (Mitchelmore & White, 2007). In general conversation, we usually do not say that a television costs 103 dollars; rather, we may say it costs one thousand dollars. According to Hazzan and Zazkis (2005), certain types of mathematical concepts are more abstract than others; and the ability to abstract is an important skill in learning and understanding mathematical concepts and describing and solving mathematical problems. There are several advantages of learning abstraction in mathematics. These include: revealing deep connections between different areas of mathematics; utilizing known results in one mathematical area to suggest conjecture in a related area; applying techniques and methods from one known area to prove results in a related area; and so on. However, the major drawback is that highly abstract mathematical concepts are more difficult to learn and require mature mathematical understanding and experience in order to be integrated with other mathematical concepts.

The field of computer science is enriched with various kinds of entities characterized as “abstract.” Computer programming involves the definition of abstract data types; and programming languages facilitate various levels of data abstraction and procedural abstraction (Colburn & Shute, 2007). Abstraction allows programmers to separate categories and concepts from instances of implementation so that there is no dependence on a particular software or hardware platform. It is a process by which data and programs are defined with a representation similar in form to its meaning (semantics), while hiding away the implementation details (Rugaber, 2006). Abstraction typically results in the reduction of a complex idea to a simpler concept or a general domain, which allows the understanding of a variety of specific scenarios in terms of certain basic ideas. According to Colburn and Shute (2007), “abstraction through information hiding is a primary process factor in computer science progress and success.” (p. 169).

Thus, the term abstraction has different meanings and understandings in different fields, such as education, psychology, mathematics, and computer science. However, there is no doubt that it is an important skill across all disciplines. Although there is no agreement for a unique meaning of abstraction, there is a unique consensus that the notion of abstraction can be taught from different perspectives. The rest of the paper explores the importance and usage of abstraction in software engineering and computer science followed by a discussion on the importance of abstraction skills in the early undergraduate and high school computer science curricula. The discussion closes with a concept of teaching abstraction skills to prospective computer science students with simple mathematical problems.

**Importance of Abstraction in Computer Programming**

Abstraction is one of the four pillars of object orientation, namely: abstraction, encapsulation, inheritance, and polymorphism (Alhir, 1998). According to Abbott and Sun (2008), all software development is abstraction; every conceptual model is an abstraction. Abstraction is similar, but different from the concept of encapsulation. In encapsulation, everything is contained in the program code, but the implementation details of a class are kept hidden from the users. On the other hand, in abstraction only the necessary amount of information is presented to the user.

It is possible to teach students how to understand and use abstraction skills, and students can be taught how to recognize some of the situations in which abstractions may be useful (Abbott & Sun, 2008). In computing, abstraction is often characterized by two major concepts: (1) abstraction by parameterization; and (2) abstraction by specification (Abbott & Sun, 2008). Abstraction by parameterization requires software abstraction, where computational elements are represented in an abstract form and parameters are only entered on the instantiation phase. This type of abstraction is important in creating elegant and generalizable programming solutions. Abstraction by specification takes place at the earlier stages of software design, such as gathering of requirements.

The removal of unnecessary detail is an important requirement in software design. Thus, abstraction is one of the fundamental principles of software engineering that determines a student’s ability in mastering complexity (Alhir, 1998) by eliciting important requirements, as well as removing unnecessary details. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time. Abstraction skills are essential in the construction of appropriate models, designs, and implications that are fit for handling a particular task at hand. Abstractions can make program code more clear and readable. According to Charles Krueger (1992), “abstraction plays a central role in software reuse.” (p. 131). In order to achieve effective reuse of software artifacts, concise and expressive abstractions are essential. Without abstraction, software developers would be forced to sift through a collection of reusable artifacts trying to figure out what each artifact did, when it could be reused, and how to reuse it. Higher level abstractions in a reuse technique reduce the effort required to go from the initial concept of a software system to representations in the reuse technique; and the lower level abstractions reduce the effort required to go from abstractions to an executable implementation. Using abstraction and decomposition of requirements are essential computational skills in designing a large and complex system.

A system can have several abstraction layers whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the computer hardware where the program is run, while high-level layers deal with the business logic of the program. In object-orientation, abstraction determines the essential characteristics of an object. It captures only those details about an object that are relevant to the current perspective. There exist several levels of abstraction in a typical object-oriented system. At the highest level, a program is viewed as a “community” of objects that interact with each other in order to achieve their common goal (Budd, 2002). The important features of this level are the lines of communication and cooperation, and the way in which the members must interact with each other. The next level of abstraction is neither found in all object-oriented programs, nor is supported by all object-oriented languages. However, many languages permit a group of objects working together to be combined into a unit. Examples of this idea are included in namespaces in C++, packages in Java, or units in Delphi. The unit allows certain names to be exposed to the world outside the unit, while other features remain hidden inside the unit. The next two levels of abstraction deal with the interactions between two individual objects. This intuition could be described by communication as an interaction between a client and a server. The next level of abstraction looks at the same boundary but from the server side. This level considers a concrete implementation of an abstract behavior. Finally, the last level of abstraction performs a specific activity, usually defined by a single method of a class. For example, the technique used to perform the addition of the most recent element placed into a stack (Budd, 2002).

Some levels are closed to the device such as the multiplication of integers. Being closed to the device means that an application is not dependent on many other levels of abstraction. Other applications (e.g., getting GPS maps on a Smartphone) are not closed to the device and are dependent on several other levels of abstraction. There can be a tradeoff between the number of levels of abstraction in a device and the efficiency of an application on the device. Too many levels of abstraction might slow down an application (Budd, 2002). On the other hand, too few levels of abstraction might result in applications that have low cohesion or take too long to create in markets where being first is important. The potential tradeoff between the number of levels of abstraction and efficiency is one of the many challenging problems in computer science.

**How Abstraction is Emphasized in Computer Science Curricula**

In the new AP Computer Science Principles (CSP) curricula (csprinciples.org), abstraction has been identified as one of the seven Big Ideas, namely: creativity, abstraction, data, algorithms, programming, Internet, and impact. In the CSP curricula, it is expected that by mastering the concept of abstraction, students will be able to: (1) describe the combination of abstractions used to represent data; (2) explain how binary sequences are used to represent digital data; (3) develop an abstraction for a specific computational need; (4) use multiple levels of abstraction in computation; and (5) use models and simulations to raise and answer questions. Moreover, the ACM/IEEE Computer Science Curricula-2013 (ACM/IEEE CSC-2013) (2013) has mentioned abstraction as a “fundamental concept in computer science.” (p. 65) According to the ACM/IEEE-CS Curricula, computer science students must learn to recognize the importance of abstraction. They need to think at multiple levels of detail and abstraction. They must learn to integrate theory and practice, to recognize the importance of abstraction, and to appreciate the value of good engineering design. The ACM/IEEE CSC-2013 gives some recognition of the importance of abstraction by mentioning its importance in the advanced level courses such as: Advanced Programming Constructs, Algorithms and Design, Development Methods Cross-Layer Communications, Graphics and Visualization, Information Management, Operating System, Memory Management, Parallel and Distributed Computing.

In CS2013, abstraction, modeling and simulation are considered essential topics for computer science. According to these curricula, computer science students need to be familiar with and able to explain the concept of modeling and the use of abstraction that allows the use of a machine to solve a problem. They need to understand several recurring themes, such as abstraction, complexity, and evolutionary change, and a set of general principles, such as sharing a common resource, security, and concurrency. “This understanding should transcend the implementation details of the various components to encompass an appreciation for the structure of computer systems and the processes involved in their construction and analysis.” (p. 21) The ACM/IEEE CSC-2013 emphasizes that any introduction to computer science would either include or presume an introduction to computing. Topics relevant to computer science include fundamental concepts in program construction, algorithm design, program testing, data representations, and basic computer architecture. The ACM/IEEE CSC-2013 also considers a general set of modeling and simulation techniques, data visualization methods, and software testing and evaluation mechanisms as important computer science fundamentals. Abstraction has been emphasized in the success of computer programming as well. According to the ACM/IEEE CSC-2013, in the introductory programming-focused CS courses students learn about concepts in computer science (e.g., abstraction, decomposition, etc.) through the explicit tasks of learning a given programming language and building software artifacts. The ACM/IEEE CSC-2013 furthermore states that:

“A principal approach to computing is to abstract the real world, create a model that can be simulated on a machine. The roots of computer science can be traced to this approach, modeling things such as trajectories of artillery shells and the modeling cryptographic protocols, both of which pushed the development of early computing systems in the early and mid-1940’s.” (p. 65).

**Teaching Abstraction Skill to the First-Learners**

In the ACM/IEEE CSC-2013 curricula, a number of courses rely on or utilize abstraction to explain, model, specify, reason or solve problems. However, none of these courses is a course on teaching abstraction. This leads to the question as to whether or not abstraction is teachable. According to the distinguished childhood psychologist Jean Piaget’s (1896–1980) theory of cognitive development, there are four possible stages of human cognitive development from infants to adulthood: sensorimotor, pre-operational, concrete operational, and formal operational stages (Kramer, 2007). The first two stages are from infancy to early childhood (at about seven years of age), where intelligence is roughly indicated by sensory motor activity and then by language and early symbol manipulation, respectively. These do not concern us, as all humans normally attain these. The third is the concrete operational stage, from about seven to twelve years of age, where intelligence is roughly indicated by a grasp of conservation of matter, of causality and an ability for classification of concrete objects. At this stage, children begin questioning to a certain degree the premises their lives have been built upon; and become able to deal with dualities, such as: things are good or bad, right or wrong, black or white. For example, a child may think that drinking alcohol is not good for parents. Finally, the formal operational is the fourth and final stage that forms around 12 years of age to adulthood, when humans hold diverse opinions about the same issues. At this stage 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 (Kramer, 2007). Further studies and experimental evidence support Piaget’s hypothesis that children progress through the first three stages of development; however, some adolescents do not progress to the formal operational stage, at all, as they mature. Tests conducted on adolescent and adult populations indicate that only 30% to 35% of adolescents achieve the formal operational stage and some adults never do (2007). Biological development may be a prerequisite for developing deep abstraction skills; and particular environmental conditions and training may be required for adolescents and adults to achieve the formal operational stage (Kramer, 2007). Huitt and Hummel (2003) recommended the following techniques: 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. These techniques may help adolescent students with low attainment rates in the formal operational stage. Thus, adolescents to early adulthood are appropriate age levels for students to learn and enhance their abstraction skills.

Another follow up question arises related to how abstraction should be taught to young students, especially the novice and prospective computer science majors. As we have seen earlier in this paper, there is no agreement for a unique meaning of abstraction; however, there is a unique consensus that the notion of abstraction can be taught in different perspectives (Hazzan & Zazkis, 2005). According to Kramer (2007), abstraction skills “must be taught indirectly through other topics.” (p. 41) Some renowned researchers (e.g., (Hazzan & Kramer, 2006), (Hill, Houle, Merritt, & Stix, 2008)) accomplished an extensive amount of research in developing an instrument to test students’ abstraction skills. Most of their research works are devoted to examining the presence of abstraction in computer science curricula and to overcome the challenges of teaching abstraction at the undergraduate level. Computer science researchers, particularly those who focus on teaching and enhance computer science at the entry level in colleges and high school, have been working to develop effective ways of enhancing their students’ ability for thinking abstractly (Hill et al., 2008).

According to Aharoni (Aharoni, 2000), computer science students possess three levels of abstraction skills that can be associated with low to high levels of abstract thinking knowledge. With low level abstraction skills, students perform Programming-Language Oriented thinking (‘action’ based); middle level abstraction skills indicate Program-Oriented Thinking (“process” based, where reference to a programming language is required, but not necessarily a specific one). Finally, with higher level abstractions, students can perform Programming-Free Thinking (“object” based). Furthermore, Hill, Houle, Merritt, and Stix (Hill et al., 2008) have identified three levels of abstraction skill: (1) conceptual abstraction skill, (2) formal abstraction skill, and (3) descriptive abstraction. Conceptual abstraction skill is the ability to acquire or form concepts that measure abstraction as it may be understood from an individual’s orientation within a big picture vs. short context. Formal abstraction skill is the ability to infer symbolic situations in order to deduce, derive, or perceive the underlying structure of a more simplified or straightforward view. Descriptive abstraction skill is the ability to discern characteristics of chief importance and, therefore, to construct generalized accounts or models that are meaningful yet abbreviated. Only humans have this kind of ability to acquire concepts. An individual with strong abstraction skills can move effectively between both formal and descriptive thinking (Hill et al., 2008). Formal abstraction and descriptive abstraction flow from conceptual abstraction, and are considered as special skill in problem solving and language manipulation, respectively (Hill et al., 2008).

**Teaching Abstraction through Mathematical Problem Solving**

According to Kramer (2007), “mathematics is an excellent vehicle for teaching abstract thinking.” (p.41) Devlin (2003) supported this experience by remarking, “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.” (p. 37) Then the question arises – what kind of mathematical knowledge is required for developing young and prospective computer science majors’ abstraction skill? Both mathematics and computer science are entirely about abstraction. Developing interaction patterns through information hiding is the primary focus of abstraction in mathematics (Colburn & Shute, 2007; Devlin, 2003). Devlin (2003) admitted the importance of discrete mathematics that has made the field of software engineering “quite unlike the other engineering disciplines, with their heavy dependency on calculus-based, continuous mathematics.” (p. 39) Devlin (2003) further stated that, from an educational point of view, in order to develop the ability to reason about formal abstractions, it is largely irrelevant exactly which abstractions are used. The main benefit of learning and doing mathematics is the fact that it develops the ability to reason analytically and precisely about formally defined abstract structures.

Although there are evidences that individuals with no mathematics education beyond high school have developed successful computer programs, it is inferred that with a more substantial mathematical background those successful individuals might have been even more successful (Devlin, 2003). If students have a lack of sufficient mathematical content knowledge, many computer science students may lack abstraction skills and face difficulty when dealing with complex problems (Kramer, 2007). In computing, it is crucial that students are able to manipulate various mathematical symbolic and numerical formalisms, as well as moving informal and complicated real-world problems to simplified abstract models.

Aharoni (Aharoni, 2000) developed a multidisciplinary cognitive model of learning computer programming to investigate the cognitive representations of students’ computational thinking. According to the model, students build a cognitive framework of a new concept of an entity by initializing some “*Actions”* performed on some physical or mental objects. When the learner gains the ability to refer to these actions using symbols, without carrying out the specific actions themselves, the actions are transformed into the next stage called “*Process”.* Finally, the *process* is transformed into an “O*bject,”* which may be used as input to new actions to be transformed to a process, then to a new - more abstract - object. In fact, the whole process is considered as one procedure of abstraction. Based on this model, abstraction knowledge can help students to develop their factual, procedural, and conceptual design knowledge iteratively and to reapply their skills creatively in other contexts.

Aharoni’s (Aharoni, 2000) Actions-Process-Object model is considered one of the central models of abstract concepts formation by Computer Science educators. This model can help to better understand how students learn computing so that educators can construct teaching-learning activities that allow their students to acquire programming skills and techniques to more rapidly and confidently (Bower, 2011).

There can be more than one abstraction hidden in a mathematical solution to a problem. For example, consider the following situation concerning the values of two art objects. An expert estimates that, if sold individually, the sum of the prices of the two objects would be only $137. However, because of keeping the objects together as a set would substantially increase the value, the two objects sold together would be close to the product of the two individual prices or $2,886. What are the individual prices? If *x* is the price of one object, the price of the other is (137 – *x*) and the equation to solve is *x*(137 − *x* ) = 2886. Multiplying through by *x*, moving 2886 to the left-hand side, and rearranging terms yields −*x*2 + 137*x* − 2886 = 0. The quadratic formula, an abstraction which is hidden from the user, can be used to solve the equation. The inputs of A = −1, B = 137, and C = −2886 should return the prices of $26 for one art object and $111 for the other.

There are also three error checking abstractions that would make the code robust:

* if ( A == 0 ) “Cannot divide by zero”
* if ( B^2 - 4 \* A \* C ) < 0 ) “Only real roots are accepted”
* if ( root\_1 <= 0 || root\_2 <= 0 ) “Roots that are negative or zero are not plausible answers to the art question”

The first error checking abstraction above would apply to all applications that use the quadratic formula. The second would apply to some. The third is specific to the art question.

**Conclusion**

In general, abstraction is a fundamental problem solving tool in mathematics and engineering. Proficiency in abstract thinking plays a critical part in the creation of models for analysis and in the production of sound engineering solutions. Because abstraction is an important skill for computing, computer science educators should be aware of the need to develop curricula that highlights the importance of abstraction. An aptitude for learning abstraction is an important characteristic of prospective CS majors, especially of high school and entry level college students. However, no standard instrument exists to measure prospective CS students’ abstraction skills. This paper was based on some recent research on abstraction as it is applied to enhance computer science education in the United States.

**Acknowledgement**

This research is sponsored in part by NSF CE21 (CNS-1240944)

**References**

References

Abbott, R., & Sun, C. (2008). Abstraction abstracted. Paper presented at the *Proceedings of the 2nd International Workshop on the Role of Abstraction in Software Engineering,* 23-30.

Aharoni, D. (2000). Cogito, ergo sum! cognitive processes of students dealing with data structures. *ACM SIGCSE Bulletin, 32*(1), 26-30.

Alhir, S. S. (1998). The object oriented paradigm. *Retrieved July, 28*, 2013. doi:<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.4054&rep=rep1&type=pdf>

Bower, M. (2011). Redesigning a web-conferencing environment to scaffold computing students' creative design processes. *Educational Technology & Society, 14*(1), 27-42.

Budd, T. A. (2002). *An introduction to object-oriented programming* (3rd ed.). Boston: MA: Addison-Wesley.

Colburn, T., & Shute, G. (2007). Abstraction in computer science. *Minds and Machines, 17*(2), 169-184.

Devlin, K. (2003). Why universities require computer science students to take math. *Communication of the ACM, 46*(9), 37-39.

Hazzan, O., & Kramer, J. (2006). Abstraction in computer science and software engineering: A pedagogical perspective. *System Design Frontier Journal, 3*(12), 1-9.

Hazzan, O., & Zazkis, R. (2005). Reducing abstraction: The case of school mathematics. *Educational Studies in Mathematics, 58*(1), 101-119.

Hill, J. H., Houle, B. J., Merritt, S. M., & Stix, A. (2008). Applying abstraction to master complexity. Paper presented at the *Proceedings of the 2nd International Workshop on the Role of Abstraction in Software Engineering,* 15-21.

Huitt, W., & Hummel, J. (2003). Piaget's theory of cognitive development. *Educational Psychology Interactive, 3*(2)

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM, 50*(4), 36-42.

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys (CSUR), 24*(2), 131-183.

Mitchelmore, M., & White, P. (2007). Abstraction in mathematics learning. *Mathematics Education Research Journal, 19*(2), 1-9.

Rosch, E., & Mervis, C. B. (1975). Family resemblances: Studies in the internal structure of categories. *Cognitive Psychology, 7*(4), 573-605.

Rugaber, S. (2006). Cataloging design abstractions. Paper presented at the *Proceedings of the 2006 International Workshop on Role of Abstraction in Software Engineering,* 11-18.

Sahami, M., Danyluk, A., Fincher, S., Fisher, K., Grossman, D., Hawthorne, B., . . . Thompson, A. (2013). Computer science curricula 2013. Retrieved July 15, Ironman Draft, from <http://ai.stanford.edu/users/sahami/CS2013/ironman-draft/cs2013-ironman-v1.0.pdf>