Knowledge, Truth, and Values in Computer Science

Timothy Colburn* Gary Shute†

Abstract

Among empirical disciplines, computer science and the engineering fields share the distinction of creating their own subject matter, raising questions about the kinds of knowledge they engender. We argue that knowledge acquisition in computer science fits models as diverse as those proposed by Piaget and Lakatos. However, contrary to natural science, the knowledge acquired by computer science is not knowledge of objective truth, but of values.

Introduction

Computer science, insofar as it is concerned with the creation of software, shares with mathematics the distinction of creating its own subject matter in the guise of formal abstractions. We have argued [1], however, that the nature of computer science abstraction lies in the modeling of interaction patterns, while the nature of mathematical abstraction lies in the modeling of inference structures. In this regard, computer science shares as much with empirical science as it does with mathematics.

But computer science and mathematics are not alone among disciplines that create their own subject matter; the engineering disciplines share this feature as well. For example, although the process of creating road bridges is certainly supported by activities involving mathematical and software modeling, the subject matter of the civil engineer is primarily the bridges themselves, and secondarily the abstractions they use to think about them.

Engineers are also concerned, as are computer scientists, with interaction patterns among aspects of the objects they study. The bridge engineer studies the interaction of forces at work on bridge superstructure. The automotive engineer studies the interaction of motions inside a motor. But the interaction patterns studied by the engineer take place in a physical environment, while those studied by the software-oriented computer scientist take place in a world of computational abstractions. Near the machine level, these interactions involve registers, memory locations, and subroutines. At a slightly higher level, these interactions involve variables, functions, and pointers. By grouping these entities into arrays, records, and structures, the interactions created can be more complex and can model real world, passive data objects like phone books, dictionaries,

*Department of Computer Science, University of Minnesota, Duluth
†Department of Computer Science, University of Minnesota, Duluth

and file cabinets. At a higher level still, the interactions can involve objects that actively communicate with one another and are as various as menus, shopping carts, and chat rooms.

So computer science shares with mathematics a concern for formal abstractions, but it parts with mathematics in being more concerned with interaction patterns and less concerned with inference structures. And computer science shares with engineering a concern for studying interaction patterns, but it parts with engineering in that the interaction patterns studied are not physical. Left out of these comparisons is the obvious one suggested by computer science's very name: what does computer science share with empirical *science*?

Metaphor and Law

We were led to this question, interestingly, when, in our study of abstraction in computer science, we found ourselves considering the role of *metaphor* in computer science [2]. Computer science abounds in physical metaphors, particularly those centering around *flow* and *motion*. Talk of flow and motion in computer science is largely metaphorical, since when you look inside of a running computer the only things moving are the cooling fan and disk drives (which are probably on the verge of becoming quaint anachronisms). Still, although bits of information do not "flow" in the way that continuous fluids do, it helps immeasurably to "pretend" as though they do, because it allows network scientists to formulate precise mathematical conditions on information throughput and to design programs and devices that exploit them. The flow metaphor is pervasive and finds its way into systems programming, as programmers find and plug "memory leaks" and fastidiously "flush" data buffers. But the flow metaphor is itself a special case of a more general metaphor of "motion" that is even more pervasive in computer science. Descriptions of the abstract worlds of computer scientists are replete with references to motion, from program jumps and exits, to exception throws and catches, to memory stores and retrievals, to control loops and branches. This is to be expected, of course, since the subject matter of computer science is *interaction* patterns.

The ubiquitous presence of motion metaphors in computer science prompted us to consider whether there is an analogue in computer science to the concern in natural science with the discovery of natural laws. I.e., if computer science is concerned with motion, albeit in a metaphorical sense, are there laws of computational motion, just as there are laws of physical motion? We concluded [3] that there are, but they are laws of programmers' own making, and therefore prescriptive, rather than descriptive in the case of natural science. These prescriptive laws are the programming invariants that programmers must first identify and then enforce in order to bring about and control computational processes so that they are predictable and correct for their purposes. The fact that these laws prescribe computational reality rather than describe natural reality is in keeping with computer science's special status, that it shares with mathematics and engineering, as creating the subject matter that it studies. This seems well and good, but it begs an obvious question: aside from the metaphors and analogies, what does computer science really have in common with science as ordinarily conceived by philosophy?

We contend that the similarity relationship between computer science and natural science is deeper than mere metaphorical language would suggest. To make the case, we

consider as background two approaches at opposite ends of a continuum of models of knowledge acquisition. At one end is the acquisition of concepts in children as studied by J. Piaget [12, 13]. At the other end is the general philosophy of science as elaborated by I. Lakatos [9, 10].

Models of Knowledge Acquisition

Piaget's work is of interest to us because he attributed the development of intelligence in children to layers of concepts embodying structural relationships, much like the arrangement of various objects in the abstraction layers employed by software designers and programmers. Piaget studied the development of concepts like number, movement, speed, causality, chance, and space. He was particularly interested in how children's primitive concepts become more sophisticated as more experience is brought to bear on them. In his words,

> ... [T]he input, the stimulus, is filtered through a structure that consists of the action-schemes (or, at a higher level, the operations of thought), which in turn are modified and enriched when the subject's behavioral repertoire is accomodated to the demands of reality. The filtering or modification of the input is called *assimilation*; the modification of internal schemes to fit reality is called *accomodation*. [13, p. 6]

The modification of internal schemes to accomodate new demands of reality can be seen to model how software designers and programmers work, as we discuss below.

At the other end of the spectrum of knowledge acquisition models are the various general philosophies of science. Consider the philosophy of Lakatos regarding what he calls "research programs", which are temporal progressions of theories and models within a science. For example, the Newtonian research program is that culminating in Newton's three laws of motion. For Lakatos, a research program contains methodological rules that both inhibit and encourage various lines of research. He calls these rules the "negative heuristic" and "positive heuristic", respectively:

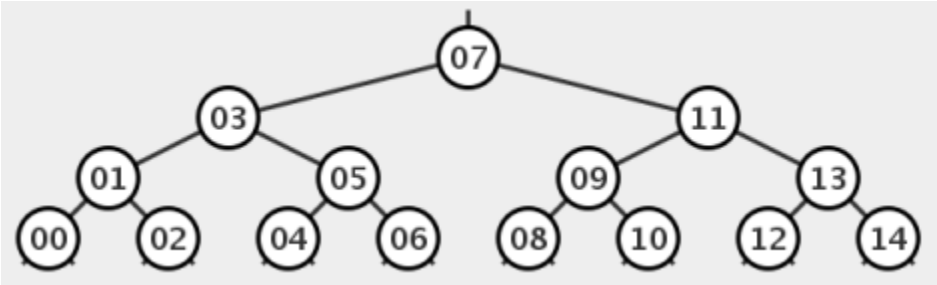
> This research policy, or order of research, is set out—in more or less detail—in the *positive heuristic* of the research programme. The negative heuristic specifies the 'hard core' of the programme which is 'irrefutable' by the methodological decision of its proponents; the positive heuristic consists of a partially articulated set of suggestions or hints on how to change, develop the 'refutable variants' of the research programme, how to modify, sophisticate, the 'refutable' protective belt. [9, p. 50]

As noted by B. Indurkhya [8], Lakatos' philosophy of science can be understood in terms of Piaget's assimilation and accomodation. The negative heuristic of a research program can be viewed as assimilative because it refuses to change a hardcore theory to fit new demands. An example within the Newtonian research program is Newton's laws of motion. The positive heuristic, on the other hand, is accomodative, allowing an adjustment of the research program's protective belt. This description of research programs is strikingly similar to Piaget's schemes ("schema" in his earlier writing [12, pp. 407–417]).

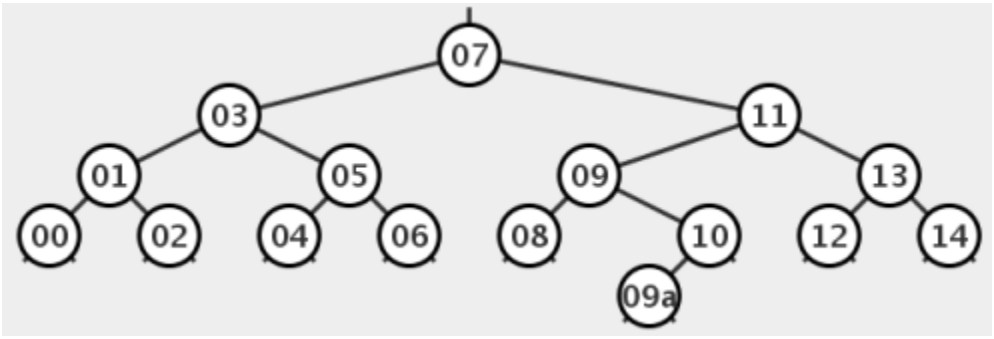
Accomodation in Computer Science

Indurkhya has pointed out the role of accomodation in the acquisition of knowledge through metaphor, and we have highlighted the importance of metaphor in computer science. But beyond that, the creation, refinement, and evolution of software structures and designs can be seen to fit both Piaget's and Lakatos' views of knowledge acquisition.

For an example, consider a data structure known as a *binary search tree* (BST). BSTs facilitate looking up data using a key. Their functionality is similar to telephone directories, where the keys are people's names and the data are addresses and telephone numbers. Here is a BST whose keys are simple strings (for simplicity, the data associated with the keys are not shown):

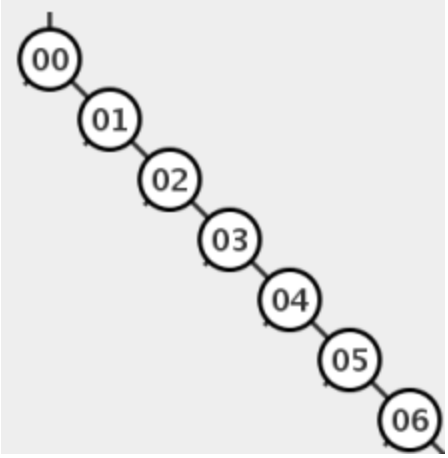


In order to preserve order among a BST's data items, a programmer must maintain the following invariant: for any node in the BST, all keys in its left subtree must be less than its key, and all keys in its right subtree must be greater than its key. When a new node is added to a BST, the following algorithm is followed: First, the node's key is compared with the key of the tree's root (the top-most node). If it is less, the new node will be placed in the left subtree, otherwise the right. The appropriate subtree is recursively searched until an available space is found on one of the tree's leaves (bottom-most nodes). Here is the example tree after the node with key **09a** is added:



This arrangement facilitates data retrieval by key, since a key can be located in time proportional to the height of the tree. If a tree is balanced, as in the one above, a key can be located efficiently even if the number of nodes is large. For example, a balanced tree of one million nodes has a height of about 20.

Unfortunately, the structure of a BST is determined by the order in which nodes are added to it, so nothing guarantees that a BST will be balanced. Here is a BST in which nodes with keys **00**, **01**, **02**, **03**, **04**, **05**, and **06** have been added in that order:



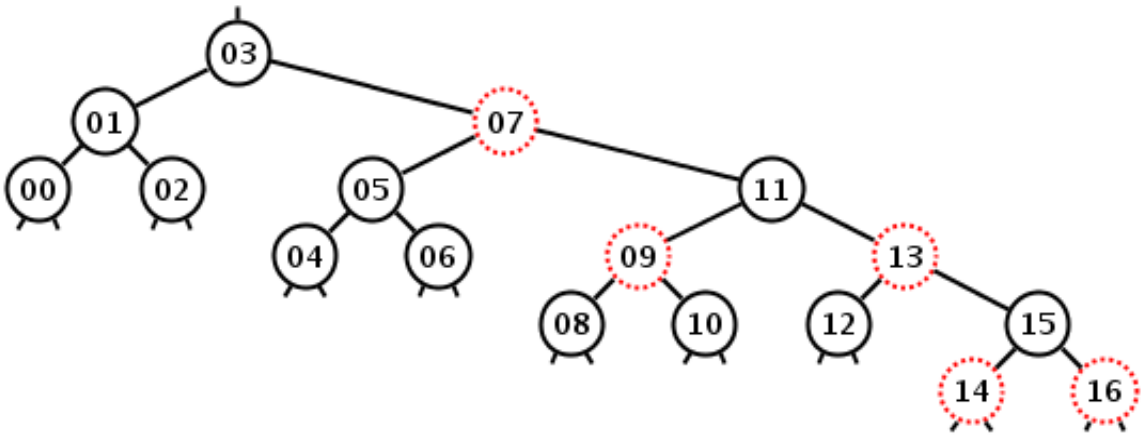
Although this structure satisfies the order invariant for BSTs, it cannot be efficiently searched since it is not balanced. If one million nodes are added to a BST in key order, finding nodes with higher numbered keys will take time proportional to its height, which is one million (compared to 20 in a balanced BST of a million nodes).

To solve this problem, computer scientists have devised a kind of self-balancing BST known as a *red-black tree* (RBT). In addition to the ordering invariant imposed on BSTs, RBTs introduce the concept of a node's *color*, requiring every node to be either red or black with the following additional constraints:

1. All downward paths from the top (root) of the tree to the bottom (leaves) must contain the same number of black nodes, and

2. The parent of a red node, if it exists, is black.

Here is a BST that is also a RBT (red nodes are also shown as dashed circles):

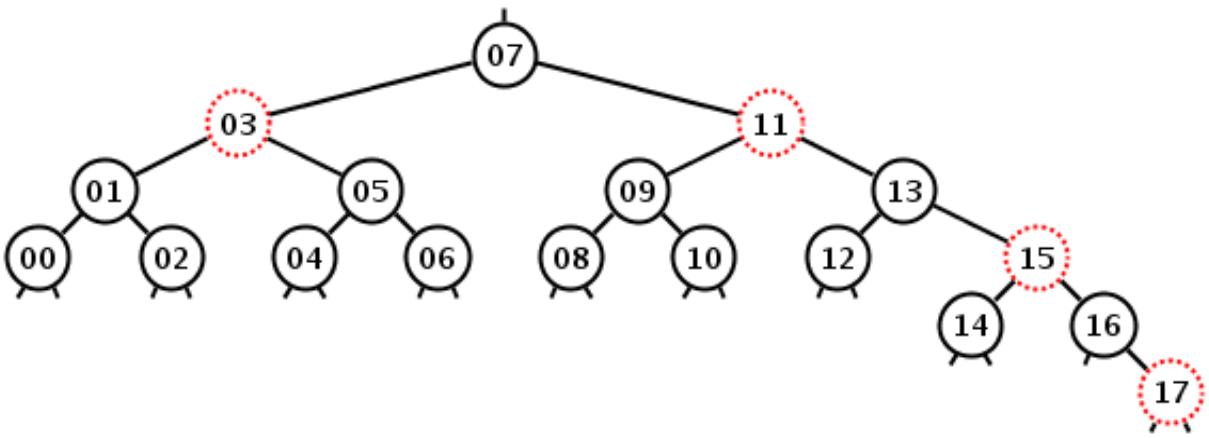


The RBT constraints do not make a search tree perfectly balanced, but they do ensure that no search path is longer than twice the length of search paths in a perfectly balanced tree. Thus in a tree with one million nodes, search paths will be no longer than 40.

This arrangement works without a hitch for some cases. Consider adding a node with key **09a**. Using the standard BST adding algorithm it can be added as the left subtree of node **10**. Then it can satisfy the RBT constraints by being colored red. Now consider

adding a node with key **17**. The BST adding algorithm would put it as the right subtree of node **16**. However, coloring it black would violate RBT constraint 1, while coloring it red would violate RBT constraint 2.

However, research into the behavior of RBTs revealed that by tweaking them in various ways (through structural changes known as rotations and certain recolorings) they can be nudged into satisfying both RBT constraints:



This is a paradigm example in computer science of structure being modified to accomodate the demands of reality. Similarly, the inviolability of the basic BST structure and its ordering invariant can be viewed as the negative heuristic in the "research program" of studying the computational representation of ordered lists of data, while the additional constraints imposed by turning BSTs into RBTs can be viewed as arising from the positive heuristic. These constraints can then become part of the negative heuristic of a new research program.

Just a Historical Reconstruction?

Some may argue that the preceding example of assimilation and accomodation in computer science suffers from the problem that we have very little evidence to support the claim that BST data structure researchers actually went through the accomodative thought processes as described. After all, researchers typically devote minimal time in their writing explaining the motivations behind the algorithms, and we can only offer *plausible* explanations of their motivating logic (the invariants). The account we have just given, therefore, is just a historical reconstruction.

But historical reconstructions have their place. Lakatos, in addition to dealing with philosophy of science, addresses issues in meta-philosophy of science, specifically criteria that can be used to assess methodology in the philosophy of science. He views historical reconstructions of science as an important part of that methodology. "... the historiography of science should learn from the philosophy of science and vice versa. It will be argued that (a) philosophy of science provides normative methodologies in terms of which the historian reconstructs 'internal history' and thereby provides a rational explanation of the growth of objective knowledge; (b) two competing methodologies can be evaluated

with the help of (normatively interpreted) history; (c) any rational reconstruction of history needs to be supplemented by an empirical (socio-psychological) 'external history'." [9, p. 102]

Lakatos gives concise, but controversial, criteria for assessing research programs in philosophy of science. These criteria capture the heart of a naturalistic philosophy of science. "Thus progress in the theory of scientific rationality is marked by discoveries of novel historical facts, by the reconstruction of a growing bulk of value-impregnated history as rational." (p. 133) In other words, we accept that scientists for the most part know what they are doing, though perhaps only intuitively, and we expect that they often will not be able to explain what they are doing. A naturalistic philosopher of science assumes that there is usually a rationality behind that intuition. For Lakatos, the task of philosophy of science is to uncover that rationality, much like the truth is uncovered by the judgments of juries in court cases. (p. 137)

Although Lakatos does not mention it, biology and paleontology, when investigating the evolution of species, are necessarily involved in historical reconstruction. Evolutionary science starts with the knowledge that traits are passed on — imperfectly — from one generation to the next. It is also known that traits that lead to higher fecundity (roughly, the number of offspring per individual) will eventually become dominant in a biological population. Fecundity is partly accounted for by success at survival and partly by success at reproduction. However, neither survival nor reproductive success can be explained in context-free terms. Traits that are successful for one species may be detrimental in another. Bigger horns or antlers may contribute to fecundity in one species but not in another. Traits interact in complex ways and in most cases must be considered in a complex context. When looking at the development of traits in a single sequence of species the evidence is often ambiguous. Evolutionary scientists just do not know enough about the lives of individual species to make convincing arguments. They can only make plausible historical reconstructions of the past. The justification for evolutionary theory lies not in individual reconstructions, but the enormous body of plausible arguments covering many different species.

Apart from the necessary historical reconstruction involved in evolution studies, some may argue that historical reconstruction in other disciplines borders on our notion of rationalization in its worst sense. But this is mitigated when we consider the importance of historical reconstruction for science and engineering education. When Lakatos looks at scientific progress, to a large extent he treats research programs as single organic wholes. But in fact, they consist of individuals who must be educated before they can participate. This gives rise to a pedagogical role for historical reconstruction. We, as scientists rather than as philosophers (or perhaps as philosopher-scientists working within the discipline), must reconstruct the developments in our science, bring out their underlying rationality, so that new generations of scientists can be "brought up to speed" more rapidly. This is even more important in a rapidly developing discipline such as computer science.

Progress in scientific disciplines (including engineering disciplines) requires that new generations "stand on the shoulders of giants" — their predecessors. But successive generations must climb higher to get to those shoulders. And they must do it in roughly the same amount of time that their predecessors took to climb to earlier shoulders. Laws are an important part of making the climb easier, even if laws as such did not govern the thinking of the earlier giants.

Consider civil engineering for example. Looking back in history it seems obvious that Minoans, Egyptians, Greeks, and Romans had a substantial body of knowledge about building roads, bridges, pyramids, aqueducts, and waste disposal systems. Much of that knowledge was probably an ill-organized body of rules for dealing with particular problems. This kind of knowledge can only be acquired through long experience. Good laws, the laws of physics for example, capture more information in a condensed form that is more readily picked up by a new generation.

We have been arguing that in computer science these laws amount to programming or software design invariants. But it is often asked rhetorically, "Why do most programmers just hammer away at the code until it works, then deal with invariants as an afterthought, if at all?" in an effort to de-emphasize the role of invariants in computer science. The answer to this question is simple: coming up with laws (or invariants) is difficult. History provides ample evidence of that. Consider the short history of civil engineering given above for example. It took two millenia to work out the laws of physics that consolidated (and made computational) the existing knowledge.

There is a stark difference between civil engineering as a discipline whose learning is based on apprenticeship and civil engineering as a discipline whose learning focuses on physical principles. For those who already have the experience, the physical principles may be just an added burden. For the next generation, though, they are a quicker way of acquiring the knowledge necessary to do the job.

With regard to invariants in computer science, it is not surprising that practicing and student software developers often use invariants as an afterthought, if at all. They are often in the position of acquiring experience without the benefit of the laws. Later, invariants are introduced as an advanced technique of algorithm development. Then students are asked not only to use them to develop and code algorithms, they are also asked to develop the invariants. It would make more sense to first give students invariants, only asking them to use them to develop algorithms, then ask them to develop invariants after they have adequate experience with using them. But this approach places rather significant intellectual demands on students first learning computer science, so some educators prefer not to use it.

Design Patterns as Research Programs

Knowledge of structures in computer science finds expression not only in the development of data structures and their algorithms but also in the discipline of object-oriented programming (OOP). OOP, the dominant style of software development today, organizes program objects into abstractions called *classes*. Identifying and relating classes in such a way that they are reusable and modifiable requires that the classes be maximally *cohesive* and minimally *coupled*. That is, objects of different classes must hide as much detail as possible from one another about how they go about implementing the concept they are intended to model. The history of OOP development has taught that software reusability and modifiability lies in decoupling objects of different classes by reducing class *dependencies*, whether those dependencies have their origin in hardware or software platform features, required object operations, required object state, or required algorithms. To the extent that classes have these kinds of dependencies on one another reduced, they need to know fewer details about each other, and they therefore exploit *information hiding*.

(See [1] for more about the role of information hiding in computer science.)

How to enforce information hiding in OOP is the objective of an approach to software development taken today through *design patterns* [6]. Design patterns first gained an audience in connection with building architecture, but the idea can be applied to software as well. Minimally, a design pattern is composed of a design problem, a context in which the problem is situated, and a design solution. Using a design pattern requires knowing when a problem and context match a given pattern and being able to carry out the given solution. For example, suppose a software developer needs to implement an inventory system for a retailer, so that when a change to the inventory occurs it is reflected immediately in all the displays throughout the store. This general problem, in which an observed subject (the inventory in this instance) needs to notify observing objects (store displays) about changes that occur to it, has been solved often enough that there is a well-known pattern, over time given the name *Observer*, that solves it. Observer accomplishes its solution elegantly by setting up the observed subject and the observing object so that required actions can be triggered in the latter when events are detected in the former without the former needing to know the nature of the actions that it causes. In this way the observer and the observed are effectively decoupled.

A design pattern can be viewed as the culmination of a Lakatosian research program in object-oriented program design, where the negative heuristic includes the list of conditions that must be met before it makes sense to apply the pattern. Design pattern identification and use to promote object decoupling comprises some of the most important aspects of knowledge in software development today.

Truth and Values

Despite the similarities that can be found between computer science and the various models of knowledge acquisition—whether at the individual (Piaget) or general scientific (Lakatos) level—any philosophy of computer science must account for "the elephant in the room", namely our oft-repeated fact that computer science creates its own subject matter. For it raises an obvious question: if science seeks knowledge about the subject matter it studies through the uncovering of truths concerning that subject matter, and computer science creates its own subject matter, what sort of truth is computer science uncovering?

Recall from the Introduction that computer science shares with engineering the feature of creating its own subject matter. It is therefore instructive to consider this question from the point of view of the engineering disciplines as well. Typically, such disciplines are directed towards large-scale design and manufacture of products through the management of design and production and also through quality control. These objectives, though important, do not exemplify scientific inquiry. Instead, we will consider that part of engineering disciplines that might be called *engineering science*.

Engineering science shares with ordinary natural science a concern for empirical discovery. However, rather than discovering laws that can explain observed events, engineering science attempts to discover *values* that support the design and production of the entity in question. For example, the choice of the type of bridge to build in a given situation depends upon values chosen among beauty, cost, scalability, and ability to withstand load and environmental forces, among others. Different situations require different

trade-offs among these values, and so different bridge types have been "discovered" (see for example [11]) that implement these trade-offs. In this regard, bridge types are basic knowledge to bridge builders in the same way that design patterns are basic knowledge to object-oriented programmers. But the knowledge is not about nature in itself, but about how to best situate created artifacts in nature, i.e. knowledge of values.

The discovery of bridge types is science because each type can be viewed as a research program. The general bridge structure defines the negative heuristic. The positive heuristic consists of ongoing accomodations of structural details to support the identified values necessary to enhance strengths and minimize weaknesses, and also take advantage of technological advances and better materials.

Similarly, the discovery of OOP design patterns is science because each pattern can be viewed as a research program. In the case of the Observer design pattern, the general context including an observed subject and an observing object defines the negative heuristic. The positive heuristic consists of the value choices made that require the chosen subject and object classes to be maximally cohesive and minimally coupled.

Computer Science and the Analysis of Knowledge

Computer science is science because it can be seen to engage in Lakatosian research programs. But computer science is engineering science because it is about the discovery of values rather than objective truth. This seems to raise a question for the philosophy of computer science, and perhaps the philosophy of any engineering science, because, traditionally, philosophers have defined knowledge as, at least, *justified true belief*, which, on the face of it, does not seem to involve the concept of values. So what kind of knowledge do computer scientists acquire?

Some background in epistemology is in order here. Since Plato, western philosophers have tried to define knowledge. In the *Theaetetus* it is agreed that knowledge is more than mere belief, but neither is knowledge to be equated merely with *true* belief, since one's irrational belief (consider a case of acute paranoia) could be serendipitously true. To count as knowledge, a true belief must have the proper kind of warrant, or justification, that disallows knowledge claims that happen to be true by chance. Such considerations have led to a standard definition of knowledge as necessarily involving *justified* true belief — a tripartite analysis — where it has been assumed that the justification required could be adequately spelled out by a sufficiently robust epistemological theory.

This analysis seems altogether consistent with the kind of knowledge pursued by the natural sciences, which are concerned with uncovering objective truths in the form of descriptive propositions about nature. Such sciences confer the status of knowledge on their beliefs only after they survive the test of justification by being arrived at through the confirmatory process of the venerable scientific method.

But we have just seen that computer science, owing to its status as an engineering science, does not seek to uncover objective truths of nature, but rather values, in the form of prescriptive laws, that aid in the control of computational processes. But knowledge need not always be knowledge of nature, and in computer science we are after knowledge of effective values — values that guide us in the construction of efficient algorithms and data structures, expressive programming languages, reliable and secure operating systems, and well-designed computer architectures. Computer science values, therefore,

are not "known" in the traditional sense of scientific discovery, but *adopted* because they work. Of course, the corpus of values making up modern computer science could be regarded as the core "knowledge" of the discipline, but it is knowledge of abstractions and principles that are of computer scientists' own making.

It is interesting, and perhaps not accidental, that the rise of computer science as a discipline has coincided in history with a split in epistemology between philosophers who embrace *internalism*, the view that a subject's knowledge is based on a justificatory relation it has with other beliefs the subject has internally; and those who opt for *externalism*, the view that knowledge can arise through processes of which a subject might not even be directly aware—unconscious cognitive processes for example. Some externalists, for example Dretske [4], go so far as to jettison the justification condition from the analysis of knowledge altogether.

Philosophers who go this route are sometimes influenced by Gettier's [5] famous counterexamples showing knowledge cannot be defined as even justified true belief. The basic idea behind the counterexamples is to concoct a situation in which a subject is adequately justified in believing a true proposition, but the proposition happens to be true by serendipity rather than by its relation to the subject's evidence, so the subject does not know the proposition at all. Suppose a dehydrated man in a desert believes there is a water hole in front of him on the basis of a heat mirage. The mirage is very convincing, and looks just like a water hole. Further, just over a hill and out of sight of the man there really is a water hole. So by a stroke of *epistemic luck* the man's belief (that there is a water hole in front of him) is true, but it should not count as knowledge, at least at the time the man made his judgment on the basis of a mirage.

Some externalists believe that Gettier-type counterexamples of this sort cannot be overcome, because thinking of knowledge as internally evidence- or justification-based is wrong-headed. Instead, knowledge can arise from one's situation in the world and from the reliable interaction of one's cognitive processes with that world. Thus, knowledge can arise without internal justification.

Externalism is sometimes associated with *naturalism*, or the general philosophical view that all facts, including true claims to knowledge, are facts about nature. So to check whether a claim to knowledge is true, one studies the world and a subject's place in it, rather than having the subject introspect on his or her internal beliefs to see if they justify what the subject claims to know. A particular variety of philosophical naturalism is ethical naturalism, or the view that moral facts are also facts about nature. An ethical naturalist wants to be able to "locate value, justice, right, wrong, and so forth in the world in the way that tables, colors, genes, temperatures, and so on can be located in the world." [7, p. 33]

Because of its dependence on prescriptive laws and values, computer science is a normative endeavor in the way that ethics is. In fact, there is a sense in which computer science is a quintessential example of a discipline that embodies naturalism, if "the world" in which we locate facts is the abstract, prescriptive law-abiding world that the computer scientist creates. For these worlds embody the values on which computer science depends. These values, in the form of program and design invariants we have described, do not just guide the computer scientists in their work; they *are* the laws of the world at hand.

References

[1] Colburn, T. & Shute, G., 2007, Abstraction in computer science, *Minds and machines: journal for artificial intelligence, philosophy, and cognitive science* 17:2, 169–184.

[2] Colburn, T. & Shute, G., 2008, Metaphor in computer science, *Journal of applied logic* 6:4, 526–533.

[3] Colburn, T. & Shute, G., 2009, Abstraction, law, and freedom in computer science, *Metaphilosophy* (forthcoming in 2010).

[4] Dretske, F., 1989, The need to know, in: M. Clay and K. Lehrer, eds., *Knowledge and skepticism*, Boulder: Westview Press.

[5] Gettier, E., 1963, Is justified true belief knowledge?, *Analysis* 23, 121–123.

[6] Gamma, E., Helm, R., Johnson, R., & Vlissides, J., 1995, *Design patterns: elements of reusable object-oriented software*, Boston: Addison-Wesley.

[7] Harman, G., 1984, Is there a single true morality, in: D. Copp and D. Zimmerman (eds.), *Morality, reason and truth. New essays on the foundation of ethics*, Totowa: Rowman and Allenheld, pp. 27–48.

[8] Indurkhya, B., 1992, *Metaphor and cognition*, Dordrecht, The Netherlands: Kluwer Academic Publishers.

[9] Lakatos, I., 1978, *The methodology of scientific programs*, Cambridge: Cambridge University Press.

[10] Lakatos, I., 1978, *Mathematics, science, and epistemology*, Cambridge: Cambridge University Press.

[11] Matsuo Bridge, 1999, **http://www.matsuo-bridge.co.jp/english/bridges/index.shtm**.

[12] Piaget, J., 1963, *The origins of intelligence in children*, New York: W. W. Norton.

[13] Piaget, J., 2000, *The psychology of the child*, New York: Basic Books.