SOFTWARE, ABSTRACTION, AND ONTOLOGY

ABSTRACT

This paper analyzes both philosophical and practical assumptions underlying claims for the dual nature of software, including software as a machine made of text, and software as a concrete abstraction. A related view of computer science as a branch of pure mathematics is analyzed through a comparative examination of the nature of abstraction in mathematics and computer science. The relationship between the concrete and the abstract in computer programs is then described by exploring a taxonomy of approaches borrowed from philosophy of mind.

Keywords: Computer science, mathematics, software, abstraction, ontology, philosophy of mind.

1. INTRODUCTION

Traditional philosophical problems concerning what kinds of things there are, and how to classify certain individual things, rarely hold more than academic interest. The outcomes of debates over the ontological status of bare particulars, for example, will not make headline news or affect the lives of non-philosophers very much. Of course, a decision to make an ontological commitment often *does* affect how we do science, as when the conjecture that over 90% of the universe is composed of unobserved "dark" matter prompts lengthy and risky experiments to capture a fleeting glimpse of a neutrino. Still, although the outcomes of these experiments hold enormous implications for cosmology, they will not affect the day-to-day lives of ordinary people whose work involves business, finance, trade, or technology.

"Software, Abstraction, and Ontology" by Timothy R. Colburn,
The Monist, vol. 82, no. 1, pp. 3–19.

However, the ontological status of computer software has emerged as an issue that may hold interest both for academic philosophers and for those involved in the formation of public policy for the international marketplace. As reported in a recent issue of *Scientific American* [Wallich (1997)], the apparent dual nature of software has sparked a debate involving free speech and the applicability of the International Traffic in Arms Regulations. A related controversy concerning policy governing intellectual property rights [Davis *et al.* (1996)] also appears to have its roots in a perceived dual nature of software.

The aim of this paper is not to propose solutions to these debates, but to bring to light and analyze the philosophical assumptions underlying them. I will describe the *prima facie* reasons for upholding software's duality, along with the ontological implications. Along the way, I will expose two misconceptions, namely that software is a machine made out of text, and a related view that computer science is merely a branch of pure mathematics. By describing the actual practice and infrastructure surrounding software creation today, I will support a view of software as a *concrete abstraction*, and try to explain what this could mean from an ontological point of view.

2. The Dual Nature of Software

At the center of one of the debates are cryptography programs. Although the philosophical questions raised by them are also raised by any computer program, they have emerged as a focus because of their value to a wide cross-section of society. Computer cryptography involves both the encryption and decryption of data that is passed along non-secure electronic communication lines, so that an eavesdropper on such a line would find only data that is unintelligible. Encryption and decryption algorithms are essential to ensure the privacy of data such as credit-card numbers used in Internet commerce, electronic funds transfers, and personal e-mail messages. These algorithms are encoded in the text of a computer programming language.

The controversy reported in [Wallich (1997)] concerns an author of a book on applied cryptography who was prohibited by the State department from exporting the book because it included as an appendix a floppy disk containing program text expressing algorithms for encryption software.

The software was deemed a national security risk, since encryption software that is strong enough could allow unfriendly governments or international terrorists to send one another encrypted messages without the eavesdropping U.S. government being able to decrypt them. On this view, a piece of strong encryption software is a dangerous machine, just like a cluster bomb or laser-guided missile. However, were the floppy disk *not* included with the book, the book *would* have been freely exportable even though the program text on the floppy disk was also printed in the book. As text on the printed page, the software was simply a product of every American's right to free speech. But as magnetic representations of the same software on a different medium, it qualified as part of a potentially dangerous mechanism and a threat to national security.

The philosophically interesting aspect of this controversy is that the two sides naturally appeal to ontological characterizations of software that appear to be inconsistent: the libertarian side wants to see software as text, and therefore freely distributable, while the regulatory side wants to see software as mechanism, and therefore subject to export restrictions.

3. Software as a Machine Made of Text

The text/mechanism dichotomy is also at the root of controversies over software copyrighting. At issue is the fairness of the practice of certain software companies quickly "cloning" the look and feel of other companies' successful products, then drastically underselling them to gain quick market share, at the expense of the original companies' large investment in research and development. In [Davis *et al.* (1996)], the authors point out how laws implementing copyright and patent policy were written at a time when copyright was only considered to apply to literary work, while patents were only considered to apply to the artifacts of technology, and these were mutually exclusive domains. But software, the authors argue, possesses a unique nature that engenders controversies when placed in this strict categorical scheme, so they propose to relax the constraint that the categories of text and mechanism are mutually exclusive:

> We have attempted to capture some of this conceptual confusion with a pseudo-paradox, pointing out that software is a machine whose medium of construction happens to be text. This captures the inseparably dual nature of software; it is inherently both functional and literary, both utilitarian and

> creative. This seems a paradox only in the context of a conceptual framework (U.S. law) that cannot conceive of such objects. [p. 23]

The important assertion here is that "software" is a machine whose medium of construction happens to be text." From a philosophical point of view this assertion raises two obvious questions: (1) How can a machine be constructed out of text? and (2) How can software, which is constructed using statements written in a formal programming language, be considered a machine?

The first question can be answered by distinguishing between physical and virtual machines. Physical machines cannot be constructed out of text, but virtual machines, which are painstaking descriptions of operations and data structures which may or may not be realized in a particular physical system, are abstract specifications describable by text. The relationship between a virtual machine and its realizations in hardware or software is similar to the relationship between a blueprint for a house and all the various ways it can be built. So when we distinguish between physical and virtual machines the first question ceases to be puzzling.

The second question, however, gets to the heart of the purported duality of software, and cannot be explained by appeal to the physical/virtual distinction. Software seems to be at once both textual and machine-like. After all, when one looks at a printout of a program one sees a lot of statements written in a formal language. But when one holds the same program on a floppy disk in one's hand, one feels the weight of a piece of a machine. But the nature of software's duality can be illuminated by understanding the distinction between software's *medium of description* and its *medium of execution.*

Strictly speaking, when one looks at printed program text, what one sees is not the software itself, but the encoding of an algorithm, or formal description of a computational process. As pointed out by [Fetzer (1993)], algorithms are logical structures akin to functions for the derivation of outputs when given inputs [p. 342]. Like a function, an algorithm is an abstract object, exemplifiable by many different textual descriptions. As Fetzer put sit, "As an effective decision procedure, an algorithm is more abstract than is a program, insofar as the same algorithm might be implemented in different forms suitable for execution by various machines by using different languages" [p. 343]. So although an algorithm may have many different concrete embodiments as text, whether they be ink on

paper or phosphors glowing in a cathode ray tube, the algorithm itself is an abstraction. The ontological status of this abstraction, as debated by, say, the opposing theories of platonic realism or constructivist idealism, is an interesting issue, but it is not germane to the point being made here.

Programs, which cause machines to execute the computational process specified by algorithms, are constructed not out of text, but out of electron charges, magnetic fields, pulses of light, etc., which are not abstractions, but to which we give an interpretation as binary numbers. The idea that software is a machine built from text gives rise to the notion that software is some kind of mysterious entity which is both abstract and possesses causal power. Fetzer, however, in analyzing the practice of formal program verification, has clearly pointed out the philosophical problems with confusing abstract and causal systems [Fetzer (1993)]. If we are puzzled by the difference between software's *medium of description*, which is formal language, and its *medium of execution*, which is circuits and semi-conducting elements, the response should not be to conflate the categories of text and mechanism into a yet more puzzling category that admits of both. It is not necessary to introduce into our ontology entities such as (non-virtual) machines constructed out of text if we study more carefully the relationship between software's medium of description and its medium of execution.

Software's medium of description is, of course, text, constructed out of many possible levels of formal language. Languages such as C++ and Java are considered *high-level* because the terms involved in them are not about machines at all, but abstract entities like variables and operations on numbers, as in the program statement `c = a + b`, which instructs a processor to add the values in locations `a` and `b` and store the result in location `c`. A programming tool called a *compiler* allows programmers to think in more abstract terms than the state of semiconducting elements, because of its ability to transform one kind of program representation into another. Typically, this other representation is *assembly language*.

One assembly language representation, among many, of the high-level statement `c = a + b` is:

```
copy  a, regl
add   b, regl
copy  regl, c
```

These statements accomplish the same task as the high-level statement, but they reveal something about a specific machine's architecture while doing so, namely, the existence of a piece of hardware called a *register* (`regl` in this example). Prior to the widespread availability of compilers, most programmers had to write in languages like this. Now, hardly any programmer does, because the creation of this type of program is automated. But while this description involves concrete objects like registers and memory locations, it still has no relation to the medium of execution, which is circuits etched in silicon. Also, it is not written in the language of a central processor, which is the binary language of ones and zeros. Another programming tool called an *assembler* transforms assembly language representations of programs into *machine language* representations, whose elements are ones and zeros. One type of assembler would translate the above assembly language instructions into the following machine language instructions:

```
00010001000100010000011110111000
10000001000100010000011110111010
00010001001100010000011110111100
```

This is the only type of program representation that a central processor "understands." It obliquely refers to operations, registers, and memory locations through a complicated encoding method called a *machine code format*. If you were a programmer in the dark days before language translators, you endured painstaking hours creating a program like this on a piece of paper, and then you had your time with the machine. You laboriously "loaded" the program into memory by flipping switches on the front panel of the machine, up or down depending on whether you were loading a zero or one. The physical effect of this was to place bi-stable memory elements, be they vacuum tubes, magnetic core drum elements, or transistors, into one of two states which, through an act of stipulative definition, could be regarded as a one or a zero. A "category shift" in the programming process between text and mechanism occurred therefore with this at once physical and intentional act of placing some material into one of two states and giving the state an interpretation as a binary digit, or *bit*.

The situation is in principle no different today. Instead of flipping switches or otherwise loading a machine language program full of bits by

hand, a program called a *link-loader* does it for you. The incalculable power and utility of tools like compilers, assemblers, and link-loaders conceal the fundamental fact that to program a machine is to put its memory into a state which when presented as input to circuits will produce more states of a desired kind. The various levels of formal language which we today use to describe these states are abstract tools to achieve this. So, the remarkable thing about software is not how it is a machine whose medium of construction is text, but how a textual description of a piece of software is involved in its own creation as states of memory elements. The development over time of tools many programmers take for granted is the reason we can do this relatively easily. Formal languages are indispensable abstractions for thinking *about* software, but there is no reason to think that software is a machine made of text.

4. Software as a Concrete Abstraction

But the role of formal language in the creation of software does introduce a question of the role of abstraction in computer science. Some accounts are unabashed that a duality of abstract *vs.* non-abstract is at the heart of the discipline. For example, an undergraduate textbook, called *Concrete Abstractions: An Introduction to Computer Science* [Hailperin *et al.* (1997)], freely switches between the poles of this duality right from the start of its preface:

> At first glance, the title of this book is an oxymoron. After all, the term *abstraction* refers to an idea or general description, divorced from actual, physical objects. On the other hand, something is concrete when it is a particular object, perhaps something that you can manipulate with your hands and look at with your eyes. Yet you often deal with concrete abstractions. Consider, for example, a word processor. When you use a word processor, you probably think that you have really entered a document into the computer and that the computer is a machine which physically manipulates the words in the document. But in actuality, when you "enter" the document, there is nothing new inside the computer—there are just different patterns of activity of electrical charges bouncing back and forth between capacitors. . . . Even the program that you call a "word processor" is an abstraction—it's the way we humans choose to talk about what is, in reality, yet more electrical charges. Yet at the same time as these abstractions such as "word processors" and "documents" are mere convenient ways of describing patterns of electrical activity, they are also *things* that we can buy, sell, copy, and use. [p. 5]

The juxtaposition of seemingly incompatible ontological categories in a "concrete abstraction" may strike philosophers as puzzling. After all, the similarly curious duality in the nature of microphenomena asserted by the Copenhagen interpretation of quantum mechanics caused philosophical tremors for physicists and philosophers of science alike. [Reichenbach (1953), Hanson (1967)] The particle and wave characterizations of light were both *physical* characterizations. By contrast, the duality inherent in a "concrete abstraction" crosses *meta*physical categories, and deserves at least as serious philosophical scrutiny as the Copenhagen interpretation. While computer scientists often enthusiastically embrace this duality, a metaphysician will view it as a puzzle to be explained. How can something, namely a computer program, be at once concrete and abstract? We will return to this question later, but first we will entertain a misconception that often arises whenever computer science is characterized as being concerned with abstractions, concrete or not. This is the persistent misconception by some that computer science is just a branch of pure mathematics.

5. Computer Science and Pure Mathematics

That this point of view is widely held by influential computer scientists is carefully pointed out by Fetzer. In [Fetzer (1993)] and [Fetzer (1991)], he analyzes the practice and claims of formal program verificationists, who are most inclined to hold this view, and he concludes that the activity of computer programming is fundamentally different from the deliberations of pure mathematics. In what follows I shall come to the same conclusion, but by examining the nature of abstraction in the two activities. This will also illuminate the stage for examining the meaning of a "concrete abstraction."

Abstraction in Mathematics

Both mathematics and computer science are marked by the introduction of abstract objects into the realm of discourse, but they differ fundamentally in the nature of these objects. I will suggest that the difference has to do with the abstraction of *form vs.* the abstraction of *content.*

Traditionally, mathematics, as a formal science, has been contrasted with the factual sciences such as physics or biology. As natural sciences,

the latter are not concerned with abstraction beyond that offered by mathematics as an analytical tool. The literature is full of strict bifurcations between the nature of formal and factual science in terms of the meanings of the statements involved in them. Carnap, for example, separates them using the analytic/synthetic distinction. [Carnap (1953)] Since analytic statements are valid according to the transformation rules of a system, and not by virtue of the truth or falsity of any other statements, then formal, mathematical, or deductive systems do not say anything empirically significant. In fact, says Carnap:

> *The formal sciences do not have any objects at all*; they are systems of auxiliary statements without objects and without content. [p. 128]

Thus, according to Carnap, the abstraction involved in mathematics is one totally away from content and toward the pure form of linguistic transformations.

Not all philosophers of mathematics agree with Carnap that mathematics has only linguistic utility for scientists, but there is agreement on the nature of mathematical abstraction being to remove the meanings of specific terms. Cohen and Nagel, for example, [Cohen (1953)] present a set of axioms for plane geometry, remove all references to points, lines, and planes, and replace them with symbols used merely as variables. They then proceed to demonstrate a number of theorems as consequences of these new axioms, showing that pure deduction in mathematics proceeds with terms that have no observational or sensory meaning. An axiom system may just *happen* to describe physical reality, but that is for experimentation in science to decide. Thus, again, a mathematical or deductive system is abstract by virtue of a complete stepping away from the content of scientific terms.

As a final example, consider Hempel's assessment of the nature of mathematics while arguing for the thesis of logicism, or the view that mathematics is a branch of logic:

> The propositions of mathematics have, therefore, the same unquestionable certainty which is typical of such propositions as "All bachelors are unmarried," but they also share the complete lack of empirical content which is associated with that certainty: The propositions of mathematics are devoid of all factual content; they convey no information whatever on any empirical subject matter. [Hempel (1953), p. 159]

In each of these accounts of mathematics, all concern for the content or subject matter of specific terms is abandoned in favor of the *form* of the deductive system. So the abstraction involved results in essentially the *elimination* of content. In computer science we will see that content is not totally abstracted away in this sense. Rather, abstraction in computer science consists in the *enlargement* of content. For computer scientists, this allows programs and machines to be reasoned about, analyzed, and ultimately efficiently implemented in physical systems. For computer users, this allows useful objects, like documents, shopping malls, and chat rooms to exist virtually in a purely electronic space.

Abstraction in Computer Science

Understanding abstraction in computer science requires understanding some of the history of software engineering and hardware development, for it tells a story of an increasing distance between programmers and the machine-oriented entities which provide the foundation of their work. This increasing distance corresponds to a concomitant increase in the reliance on abstract views of the entities with which the discipline is fundamentally concerned. These entities include machine instructions, machine-oriented processes, and machine-oriented data types. I will now try to explain the role of abstraction with regard to these kinds of entities.

Language Abstraction

At the grossest physical level, a computer process is a series of changes in the state of a machine, where each state is described by the presence or absence of electrical charges in memory and processor elements. But as we have seen, programmers need not be directly concerned with machine states so described, because they can make use of software development tools which allow them to think in other terms. High-level language programs allow machine processes to be described without reference to any particular machine. Thus, specific language content has not been eliminated, as in mathematical or deductive systems, but replaced by abstract descriptions with more expressive power.

Procedural Abstraction

Abstraction of language is but one example of what can be considered the attempt to enlarge the content of what is programmed about.

Consider also the practice of *procedural abstraction* that arose with the introduction of high-level languages. Along with the ability to speak about abstract entities like statements and variables, high-level languages introduced the idea of *modularity*, according to which arbitrary blocks of statements gathered into *procedures* could assume the status of statements themselves. For example, consider the following high-level language statements:

```
for i = 1 to n do
    for j = 1 to m do
        read(A[i,j]);
for j = 1 to m do
    for k = 1 to p do
        read(B[j,k]);
for i = 1 to n do
    for k = 1 to p do
        C[i,k] = 0;
        for j = 1 to m do
            C[i,k] = C[i,k] + A[i,j] * B[j,k]
```

It would take a studied eye to recognize that these statements describe a process of filling an $n \times m$ matrix A and an $m \times p$ matrix B with numbers and multiplying them, putting the result in an $n \times p$ matrix C such that $C_{i,k} = \Sigma_{j=1}^{m} A_{i,j} B_{j,k}$. But by abstracting out the three major operations in this process and giving them procedure names, the program can be written at a higher, and more readable level as:

```
ReadMatrix (A,n,m);
ReadMatrix (B,m,p);
MultiplyMatrices (A,B,C,n,m,p)
```

These three statements convey the same information about the overall process, but with less detail. No mention is made, say, of the order in which matrix elements are filled, or indeed of matrix subscripts at all. Of course, the details of how the lower level procedures perform their actions must be given in their definitions, but the point is that these definitions can be strictly separated from the processes which call upon them. What we have, then, is the total abstraction of a procedure's use from its definition.

Where in the language example we had the abstraction of the content of computer instructions, here we have the abstraction of the content of whole computational procedures. And again, the abstraction step does not eliminate content in favor of form as in mathematics; it renders the content more expressive.

Data Abstraction

As a last example, consider the programmer's practice of *data abstraction*. Machine-oriented data types, like integers, arrays, floating point numbers, and characters, are, of course, themselves abstractions placed on the physical states of memory elements interpreted as binary numbers. They are, however, intimately tied to particular machine architectures in that there are machine instructions specifically designed to operate on them. They are also built into the terminology of all useful high-level languages. But this terminology turns out to be extremely impoverished if the kinds of things in the world being programmed about include, as most current software applications do, objects like customers, recipes, flight plans, or chat rooms.

The practice of data abstraction is the specification of objects such as these and all operations that can be performed on them, without reference to the details of their implementation in terms of other data types. The specification and construction of such objects, called *abstract data types*, are primary topics in undergraduate computer science curricula, as evidenced by the many textbooks devoted to these topics. (See [Dale (1996), Carrano (1995), Bergin (1994), Hailperin *et al.* (1997)] for some of them.) But this again is a type of abstraction which does not eliminate empirical content, as in mathematics, but rather enlarges the content of terms by bringing them to bear directly on things in a non-machine oriented world.

Abstraction in computer science therefore serves to enhance the programming process by distancing the programmer from the drudgery of concerning herself with machine processes and data types. As such, abstraction is a tool used in the construction of an artifact, namely a computer program. This program, being encoded in physical memory elements, is concrete, but its description, in text at any level, is abstract. Hence the term 'concrete abstraction'. We now return to the metaphysical question:

what is the relationship between the concrete and abstract in a computer program?

6. The Ontological Status of Concrete Abstractions

This question brings to mind one of similar form which is also motivated by *prima facie* conflicting kinds of entities. This is the question concerning the mind/body problem: What is the relationship between the physical and mental in a person? I introduce this question into the current discussion not because of a supposed analogy between programs and persons, but because of the methodological framework employed by philosophers to approach the mind/body problem. I believe that the taxonomy of solutions in terms of monism and dualism provides a useful metaphor, if not an entirely appropriate language, for describing programs as concrete abstractions.

Monism

Monism holds that the metaphysical duality apparent in talk about persons in terms of the mental and the physical is an illusion, and that a person is but one kind of entity. A variety of monism called the *double-aspect theory* asserts that there is but one kind of entity, and the mental and the physical are simply *aspects* of this kind of entity. This theory is largely discredited because of problems with describing both this other kind of entity and what exactly an *aspect* is. [Shaffer (1967)] A double-aspect theory for computer programs would suffer structurally from the same kinds of objections. In fact, the incorrect view expressed above in [Davis *et al.* (1996)] that software is a machine made of text is a kind of double-aspect theory of computer programs.

In the other kinds of metaphysical monism, namely materialism and idealism, one of the two apparent kinds of entity involved in the duality of persons is embraced as "real" while the other is not. Or, talk about one of the two kinds of entity can be eliminated, theoretically if not as a practical matter, in favor of talk about the other. Such attempts at reduction, for example, the identity theory in philosophy of mind, or phenomenalism in epistemology, often encounter difficulty when trying to translate talk about one kind of entity (for example, physical objects) into talk about a wholly different kind of entity (for example, sense reports). A reduction of

statements of formal programming language, even zeroes and ones, to statements about physical states of memory elements is even more difficult, precisely because the characterization of physical state in machine language as zeros and ones is *itself* an abstraction; the kinds of physical state that this language "abstracts" are limitless. They may be the electronic states of semiconductors, or the states of polarization of optical media, or the states of punched paper tape, or the position of gears in Babbage's 18th-century analytical engine, etc.

It is worth pointing out, however, that on the abstract side of the abstract/concrete dichotomy lies one of the most successful reductivist projects ever undertaken. The whole history of computer science has seen the careful construction of layer upon layer of distancing abstractions upon the basic foundation of zeros and ones. Each time a programmer writes and executes a high-level program, these layers are stripped away one by one in elaborate translations from talk of, say, chat rooms, to talk of windows, to talk of matrices, to talk of variables, registers, and memory addresses to, finally, zeros and ones. As an abstraction, this translation is complete and flawless. The concrete embodiment of the zeros and ones in physical state, however, is not an abstract process, but an intentional act of stipulative definition.

Dualism

Since double-aspect or reductivist attempts to characterize the abstract/concrete dichotomy for computer programs seem destined to fail, we turn to dualism. Within the mind/body problem methodology, dualism asserts that persons are composed of both mental and physical substances, or described by both mental and physical events, accounting for their having both mental and physical properties. The dualistic analogue for the computer science problem is that programs are both abstract and concrete; more specifically, it does not say what the *relation* is between the abstract and the concrete. For the mind/body problem, elaborations of dualism emerge in the form of dualistic interactionism, epiphenomenalism, and parallelism.

Dualistic interactionism posits a causal relation between the physical and the mental in both directions. So burned flesh causes a feeling of pain, and a feeling of pain causes removal of the hand from the fire. Epiphenomenalism allows causation in one direction only: burned flesh causes a feeling of pain, but the removal of the hand from the fire is caused by

certain neuro-physiological events. The feeling of pain has no causal efficacy, but it exists as an "ontological parasite" on the burned flesh. It should be clear, however, that neither of these accounts is possible with respect to the abstract and the concrete in computer programs. As Fetzer has pointed out, there can be no causal power exerted in either direction between an abstract, formal system and a physical system. [Fetzer (1993)] For the mind/body problem, specifying the causal relation between the two sides is merely problematic; for an explanation of concrete abstractions it is in principle impossible.

Pre-Established Harmony

That leaves a dualistic account of the concrete *vs.* the abstract characterizations of programs without any causal interaction between them. For the mind/body problem, this would be called a parallelistic account, and it would be mentioned only as a historical curiosity. The reason it is not entertained much today is that, without the benefit of a causal relation between mental and physical events, either (1) the observed correlation constantly being affirmed by science between the mental and physical is just a random coincidence, or (2) this correlation is deliberately brought about through a "pre-established harmony," presumably by God. Both of these alternatives are difficult to justify.

But the pre-established harmony thesis is well suited for explaining the high correlation between computational processes described abstractly in formal language and machine processes bouncing electrons around in a semiconducting medium. For, of course, it is not necessary to appeal to God in accounting for this correlation; it has been deliberately brought about through years of co-operative design of both hardware processors and language translators. An analogy used to describe parallelism in the mind/body problem imagines two clocks set and wound by God to tick in perfect synchrony forever. For the abstract/concrete problem we can replace God by the programmer who, on the one hand, by his casting of an algorithm in program text, describes a world of multiplying matrices, or resizing windows, or even processor registers; but on the other hand, by his act of typing, compiling, assembling, and link-loading, he causes a sequence of physical state changes that electronically mirrors his abstract world.

The "parallel" nature of the abstract and the concrete is a defining characteristic of the digital age. From the notion of cyberspace, to online

shopping malls, to virtual communities, worlds exist with no impact outside their boundaries. The same is true of the world of the programmer. Programmers today can live almost exclusively in the abstract realm of their software descriptions, but since their creations have parallel lives as bouncing electrons, theirs is a world of concrete abstractions.

Timothy R. Colburn

Department of Computer Science
University of Minnesota at Duluth

REFERENCES

[Bergin (1994)] Bergin, J. (1994), *Data Abstraction: The Object-Oriented Approach Using C++*, McGraw Hill.

[Carnap (1953)] Carnap, R. (1953), "Formal and Factual Science," *Readings in the Philosophy of Science*, H. Feigl and M. Brodbeck, eds., Appleton-Century-Crofts.

[Carrano (1995)], *Data Abstraction and Problem Solving with C++: Walls and Mirrors*, Benjamin Cummings.

[Cohen (1953)] Cohen, M. and Nagel, E. (1953), "The Nature of a Logical or Mathematical System," *Readings in the Philosophy of Science*, H. Feigl and M. Brodbeck, eds., Appleton-Century-Crofts.

[Cormen *et al.* (1990)] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990), "The RSA Public-Key Cryptosystem," *Introduction to Algorithms*, M.I.T. Press, 831–36.

[Dale (1996)] Dale, N. and Walker, H. M. (1996), *Abstract Data Types: Specifications, Implementations, and Applications*, Heath.

[Davis *et al.* (1996)] Davis, R., Samuelson, P., Kapor, M., and Reichman, J. (1996), "A New View of Intellectual Property and Software," *Communications of the ACM* 39(3): 21–30.

[Fetzer (1993)] Fetzer, J. (1993), "Program Verification: The Very idea," *Program Verification: Fundamental Issues in Computer Science*, T. Colburn, J. Fetzer, and T. Rankin, eds., Kluwer Academic Publishers, 321–58. The paper originally appeared in *Communications of the ACM* 31(9): 1048–63.

[Fetzer (1991)] Fetzer, J. (1991), "Philosophical Aspects of Program Verification," *Minds and Machines* 1(2): 197–216. Reprinted in *Program Verification: Fundamental Issues in Computer Science*, T. Colburn, J. Fetzer, and T. Rankin, eds., Kluwer Academic Publishers, 1993.

[Hailperin *et al.* (1999)] Hailperin, M., Kaiser, B., and Knight, K., (1999), *Concrete Abstractions: An Introduction to Computer Science*, PWS Publishing, in press.

[Hanson (1967)] Hanson, N. R. (1967), "Philosophical Implications of Quantum Mechanics," *The Encyclopedia of Philosophy*, volume 7, MacMillan, 41–48.

[Hempel (1953)] Hempel, C. (1953), "On the Nature of Mathematical Truth," *Readings in the Philosophy of Science*, H. Feigl and M. Brodbeck, eds., Appleton-Century-Crofts.

[Reichenbach (1953)] Reichenbach, H. (1953), "The Principle of Anomaly in Quantum Mechanics," *Readings in the Philosophy of Science*, H. Feigl and M. Brodbeck, eds., Appleton-Century-Crofts.

[Shaffer (1967)] Shaffer, J. (1967), "Mind-Body Problem," *The Encyclopedia of Philosophy*, vol. 5, MacMillan, Inc., 336–46.

[Wallich (1997)] Wallich, P. (1997), "Cracking the U.S. Code," *Scientific American* April 1997: 42.