Abstraction in Computer Science

Article *in* Minds and Machines · August 2007

DOI: 10.1007/s11023-007-9061-7 · Source: DBLP

2 authors, including:

Timothy Colburn
University of Minnesota Duluth
18 PUBLICATIONS **448** CITATIONS

SEE PROFILE

Minds & Machines (2007) 17:169–184
DOI 10.1007/s11023-007-9061-7

Abstraction in Computer Science

Timothy Colburn · Gary Shute

Published online: 5 June 2007

Abstract We characterize abstraction in computer science by first comparing the fundamental nature of computer science with that of its cousin mathematics. We consider their primary products, use of formalism, and abstraction objectives, and find that the two disciplines are sharply distinguished. Mathematics, being primarily concerned with developing inference structures, has *information neglect* as its abstraction objective. Computer science, being primarily concerned with developing interaction patterns, has *information hiding* as its abstraction objective. We show that abstraction through information hiding is a primary factor in computer science progress and success through an examination of the ubiquitous role of information hiding in programming languages, operating systems, network architecture, and design patterns.

Keywords Abstraction · Computer science · Information hiding · Mathematics

Introduction

Computer science is rich with references to various entities characterized as *abstract* and various activities characterized as *abstraction*. Programming, for example, involves the definition of *abstract data types*. Programming languages facilitate varying levels of *data abstraction* and *procedural abstraction*. Language architects specify *abstract machines*. One computer science textbook even characterizes its subject matter as the science of *concrete abstractions* (Hailperin et al. 1999). A philosophy of computer science should be able to characterize abstraction as it

T. Colburn (✉) · G. Shute
Department of Computer Science, University of Minnesota, Duluth, MN, USA
e-mail: tcolburn@d.umn.edu

G. Shute
e-mail: gshute@d.umn.edu

occurs in computer science, and also to relate it to abstraction as it occurs in its companion, mathematics. In this paper we attempt to do just that.

All legitimate sciences build and study mathematical models of their subject matter. These models are usually intended to facilitate the acquisition of knowledge about an underlying concrete reality. Their role is one of supporting scientific reasoning in an attempt to discover explanations of observed phenomena.

The nonmathematical models of empirical science are the arrangements of experimental apparatus by which hypotheses concerning the nature of physical, or perhaps social, reality are tested in an effort to uncover explanatory or descriptive laws of nature. In physics, for example, scientists might build high-energy particle accelerators to model conditions in the early universe, with the results of collisions of subatomic particles being used to determine the number of neutrino families there are.

Traditional empirical sciences are thus concerned with both concrete models in the form of experimental apparatus, and abstract models in the form of mathematics. Computer science, insofar as it is concerned with software, is distinguished from the empirical sciences in that none of its models are physically concrete—they are realized in software, and in this nonphysical sense computer science models are abstractions. Since mathematics is also distinguished from the empirical sciences in that the types of models it builds and studies are abstractions, one may be tempted to infer that there is a close relationship between computer science and mathematics, even to infer that computer science is a subspecies of mathematics. In fact, a tradition as old as computer science itself (see Colburn et al. 1993, Parts I and II, and the Epilogue) claims just that. In this paper, we show that this is only a surface similarity, and that the fundamental nature of abstraction in computer science is quite different from that in mathematics. In doing so, we appeal to the ways in which abstraction actually facilitates the creation of software.

It is not our intention to characterize mathematics and computer science by drawing a boundary that divides them. This cannot be done because both disciplines can be applied to a broad range of subject matter, including each other. Inevitably, this results in significant areas of overlap, with no clear-cut boundary between the two disciplines. However, we do attempt to characterize the *direction* of differences between the two disciplines, particularly with regard to their use of abstraction. Before turning to a consideration of these uses, it is informative to give some background on previous philosophical treatment of abstract entities and abstraction as a process.

Some Philosophical Background on Abstraction

Modern Western philosophical treatments of the concept of an *abstract idea* arguably began with Locke (1706), who wondered how, if our experience is of particular things, such as Lake Superior or the dripping faucet in one's home, we come by general terms such as ''water''. Locke answered this by proposing that general terms stand for abstract ideas, and that abstract ideas are created through a process of *abstraction*, which separates these ideas from spatial or temporal

qualities of particular things. We can have an abstract idea of water that does not include in it any particular idea of this or that lake, or this or that sea, or this or that dripping faucet, etc. So abstraction in this sense concerns the distinction between the general and the particular in the world of ideas. Importantly, this abstraction proceeds by dropping (or ignoring, or neglecting) spatial and temporal features from ideas.

A theory of abstract ideas may or may not be associated with a theory of *universals*, or things that can be shared by many particular things. Plato's theory of forms, for example, held that there is a world inhabited by things that can be regarded as the shared common nature of particular things in the world of experience. Justice, for example, is that which is shared by all just acts. Just acts are particular and perceptible, while justice itself is not, though it is just as ''real''. Justice is an example of an immutable Platonic *form*. Although Plato also used the term ''idea'' to refer to forms, modern language uses ''universal'' to distinguish the essence of a form from the minds that may come in contact with it.

A theory of universals may be regarded as the ontological counterpart to a theory of abstract ideas. Just as one can distinguish abstract ideas from nonabstract ideas by appealing to general terms and their relation to particular terms, one can distinguish abstract objects (apart from any ideas of them) from concrete objects. This distinction between abstract and concrete objects can play out in modern treatments of modal logic, for example the logic of counterfactual conditionals, or conditionals of the form, ''If it were the case that p, then it would be the case that q''. Lewis (1973), for example, bases his logic of counterfactual conditionals on a similarity ordering of *possible worlds*: a counterfactual conditional is true in this world (in which p is false) if every world which is sufficiently similar to this one but in which p is true is also a world in which q is true. So the statement ''If kangaroos had no tails, they would fall over'' means, for Lewis, something like, ''In all worlds sufficiently similar to this one in which kangaroos have no tails, they fall over''. Such an analysis begs for a characterization of these other, possible but not actual, worlds. Are they concrete, like this one, or are they abstract? If abstract, what exactly is the distinction between concrete and abstract?

As Lewis points out, not only is there no universal agreement on this distinction, but various ways of characterizing the distinction can conflict with one another. For example, saying that abstract objects have no spatiotemporal location and do not enter into causal interaction conflicts with the idea that *sets* are abstract, since ''a set of located things *does* seem to have a location, though perhaps a divided location'' (Lewis 1986, p. 83). And regarding the absence of causal interaction, Lewis asks why can't a cause have a set of effects?

A better way of characterizing abstract and concrete entities, for Lewis, is by looking at the process by which an abstract entity is constructed out of concrete ones, which is a process of ''somehow subtracting specificity, so that an incomplete description of the original concrete entity would be a complete description of the abstraction'' (pp. 84–85). In performing an abstraction, ''[w]e are ignoring some of its features, not introducing some new thing from which those features are absent'' (p. 86). So again, abstraction as a process is about ignoring (or dropping, or neglecting) features with which one is not concerned.

Abstraction, as a process with philosophical interest, is also considered in treatments of nonmodal logic. For example, related to axiomatic set theory, there are various versions of the *axiom of abstraction*. The *unlimited* version of this axiom states that, given any property, we can form the set of all elements having that property. Here the process is not one of moving from concrete to abstract, or from particulars to general, but from particulars with a property to a set. This axiom leads to well-known contradictions such as Russell's paradox (Russell 1903). Set theorists replaced it with a *limited* version stating that, given any property P and any set x, we can form the set of all elements of x having P. By focusing on any property we like, we can, once again, completely ignore (or neglect) any other properties that do not concern us. This principle is essential to the methodology of abstraction employed by mathematicians, which we treat in the next section.

In summary, abstraction has been characterized in philosophy, mathematics, and logic as the process of eliminating specificity by ignoring certain features. We will show that abstraction in computer science is fundamentally different by first focusing on a comparison of mathematics and computer science. In the next section we describe the fundamental natures of mathematics and computer science in terms of their primary product, use of formalism, and abstraction objectives. Treatment of the latter introduces a key component of abstraction in computer science, namely *information hiding*, which we describe in the last section.

The Fundamental Natures of Mathematics and Computer Science

We can compare the nature of mathematics and computer science by comparing their primary products. We argue that the primary product of mathematics is *inference structures*, while the primary product of computer science is *interaction patterns*. This is a crucial difference, and it shapes their use of formalism and the kind of abstraction used in the two disciplines.

Their Primary Products

There is of course a long tradition in the philosophy of mathematics, but whether the treatment is traditional (Kline 1972; Hardy 1967) or more along the lines of the modern structuralists (Benacerraf 1965; Resnik 1997; Shapiro 1997), it is clear that the production of theorems and their proofs is the central activity of mathematics. These theorems and proofs are best understood not in isolation, but as parts of larger structures.

Group theory is a good example. Mathematicians begin with a definition of (axioms for) the concept of a group, that is, a set G for which an associative binary operation * has been defined where there is an identity element for * in G and all elements of G have inverses in G with respect to *. Then they develop theorems that are true for all groups, and they identify properties that some, but not all groups have that allow further inferences. They also define concepts (like conjugacy) that can be used to infer further properties of groups and their elements. Finally, they

develop tools (like matrix representations) that help them recognize groups and allow results from other mathematical inference structures to be applied to group theory. The main objective is to be able to reason in the abstract about groups and their elements.

Group theory, viewed as an inference structure, is a collection of deductive arguments, with places in the arguments for the set of elements and the binary operation. If the hypotheses of the arguments are satisfied when terms from a non-mathematical realm are substituted in the places, then the conclusions of the arguments, with similar substitutions, can be asserted.

As we mentioned before, there is plenty of overlap between mathematics and computer science, and this overlap can include theorem producing as a role for computer science—for example, when reasoning about formal languages and the automata that process them. However, the central activity of computer science is the production of software, and this activity is characterized primarily not by the creation and exploitation of inference structures, but by the modeling of *interaction patterns*. The kind of interaction involved depends upon the level of abstraction used to describe programs. At a basic level, software prescribes the interacting of a certain part of computer memory, namely the program itself, and another part of memory, called the program data, through explicit instructions carried out by a processor. At a different level, software embodies algorithms that prescribe interactions among subroutines, which are cooperating pieces of programs. At a still different level, every software system is an interaction of computational processes. Today's extremely complex software is possible only through abstraction levels that leave machine-oriented concepts behind. Still, these levels are used to describe interaction patterns, whether they be between software objects or between a user and a system.

For example, *abstract data types* (ADTs) are encapsulations of data with the operations that can be performed on the data, such encapsulation enabling the convenient interaction of a user (human or otherwise) with the data. Creating ADTs is an example of what computer scientists call *data abstraction*. Consider the *table* ADT taught in introductory computer science courses (see for example Carrano 2007). It is an interaction pattern whose origin lies outside computer science in human interactions with telephone books and dictionaries. The heart of the abstraction is the use of a key (a person's name or a word) to access other information (a person's address or telephone number or a word's meaning). It is distinguished from other access abstractions in that the order of access in a table is determined by its user whereas abstractions such as streams, stacks, queues, and priority queues provide access in an order determined by the abstraction.

The concept of an ADT is carried further with the concept of an *object*, which is central to object-oriented programming, the current dominant programming paradigm in computer science. ADTs are passive entities in that they can only be acted upon by a controlling program as though they were glorified machine data types. Objects still encapsulate data and operations, but, as opposed to ADTs, objects are *active* entities in the sense that they can be called upon to perform operations on themselves, and they can invoke operations on other objects by passing messages to them. Thus the user/ADT interaction pattern is replaced by the

more general object/object interaction. Objects, being active data-operation combinations, are best thought of as small *virtual* computers themselves rendered by software, and an executing program is best thought of as the interaction of many objects. (The concept of virtuality is a central example of abstraction in computer science which we consider in more detail in ''Abstraction Through Information Hiding''.)

Every software system, whether it is an operating system, a networked system distributed over multiple machines, or a single user program, is a carefully orchestrated interaction of entities variously called *tasks*, *threads*, or *processes*. Each such entity is an instance of an executing program. (We will also say more about these in ''Abstraction Through Information Hiding''.) These entities compete or cooperate depending on the environment. When processes communicate over a network, they must follow a *protocol*, which is a set of rules explicitly designed for interaction.

The interaction patterns that are at once the most critical from a usability point of view and the most difficult to model using abstraction tools are those interaction patterns involving the human element. This means that there are aspects of computer science that cannot be ''abstracted away'' to make them cleaner, as is done in mathematics. This point is all too often ignored by those who emphasize a mathematical paradigm for computer science. User interface aspects of programs, because they are unpredictable and subject to exceptional behavior, are sometimes characterized as second-class interaction patterns when compared to the interaction patterns of ''more interesting'' or ''cleaner'' internal algorithms and data structures, which are much more amenable to mathematical analysis. As we remarked in the Introduction and will elaborate below, abstraction in mathematics often involves a kind of ''information neglect'' that conveniently ignores aspects of its subject matter that are considered irrelevant. Unfortunately, in most cases it is not possible to consider user interaction with software irrelevant, and the unpredictability of such interaction is a fact of life for the software modeler. As a corollary, a ''pure'' computer science that analyses only clean, human-free interaction patterns is impoverished at best.

The difference between interaction patterns for computer science and inference structures for mathematics is reflected in Fetzer's distinction between causal systems and abstract models of them. As Fetzer (1988) pointed out, the practice of formal program verification at best guarantees the correctness of algorithms, and not the correctness of actual running programs that attempt to instantiate those algorithms. While computer scientists are often concerned with reasoning about their algorithms in a formal, mathematical manner, they are in practice more often concerned with bringing about the interactions that their algorithms specify.

So abstraction in mathematics facilitates inference, while abstraction in computer science facilitates the modeling of interaction. The difference between these activities can also be seen by considering the respective disciplines' use of formalism.

Their Use of Formalism

The formalism of mathematics is relatively *monolithic*, based on set theory and predicate calculus. It is even common in mathematics to reduce as far as possible the types of formal elements used in analysis. Whitehead and Russell (1910), for example, carried the idea to an extreme, reducing counting numbers to sets of sets, integers to equivalence classes of pairs of counting numbers, rational numbers to equivalence classes of pairs of integers, and real numbers to Dedekind cuts (sets) of rational numbers. There are some exceptions. Algebraic topologists and ring theorists, for example, make heavy use of commutative diagrams, that is, graphs whose vertices are algebraic structures—groups or rings—and whose directed edges represent functions from one of the algebraic structures to another. But for the most part the ''universal language'' of mathematics is so because of a relatively small and stable collection of formal representation tools that do not change much over time and space.

Formal representation of this sort certainly occurs in computer science. In the theory of computation, for example, the object of study is not any particular computing machine, but an abstraction of all such machines. The usual presentations of this abstraction are similar to mathematical abstractions, using sets, ordered tuples, and functions. Computation is a particularly mathematical concept, and studying it will naturally involve mathematical methods.

But we have just seen that computer science, despite its name, is as much about interaction patterns as it is about computation. The myriad kinds of such patterns, as described above, and the injection into them of the human element, leads to a formalism of computer science that is *pluralistic and multilayered*, involving multiple programming languages, alternative software design notations, and diverse system interaction patterns embodied by language compilers, operating systems, and networks. Mathematics, for better or worse, has to a large extent adapted its inference structures to its formalism. The Turing machine formalism is an example where computer science has had the luxury of following mathematical formalism more closely, but it is an exception. Overall, computer science does not have that luxury as long as the production of human-centered software is its primary focus.

Moreover, changes in technology drive changes in computer science ontology, with the result being that the kinds of things that make up the interaction patterns are constantly changing. The formal languages available to software modelers began with entities like registers and addresses, moved on to variables and subroutines, then procedures and ADTs, now objects and threads. The ontology changes because the underlying technology increases speeds and decreases sizes so much that new generations of machine allow old structures to be subsumed by higher abstractions. There is of course no such influence at work in mathematics.

Their Abstraction Objectives

Among philosophers of mathematics there is agreement on the nature of mathematical abstraction being to remove the meaning of specific scientific terms. According to Cohen and Nagel (1953), for example, ''... a system is deductive not in

virtue of the special meaning of its terms, but in virtue of the universal relations between them. The specific quality of the things which the terms denote do not, as such, play any part in the system'' (p. 138). As all students of logic learn, for example, it is the *form* of modus ponens that makes it valid, and not the meanings of the sentences that comprise it. So mathematical abstraction in this sense eschews any nonlogical meaning.

We can characterize this sort of formal mathematical abstraction as an aid to discovering universal properties or relationships that hold within any individual member of a broad class of entity (for example, groups). The same sort of abstraction is employed when deducing the relationship between the lengths of the sides of any right triangle, as in the Pythagorean theorem. But another kind of abstraction is employed when deciding that certain properties, for example the color of a right triangle, or the processing speed of a computing device, are irrelevant when reasoning about the properties of the class of such things as a whole. This kind of abstraction makes judgments about what is essential to a concept and what is not. In the mathematical theory of groups, for example, all that is chosen for study is the set G and operation * with the particular properties of interest. All other concrete properties of groups are ignored. They are inessential details when considering, for example, whether two groups are isomorphic. So there are at least two kinds of abstraction in mathematics: the emphasis of form over content, and the neglect of certain features in favor of others. As Cohen and Nagel put it, ''A deductive system is therefore doubly abstract: it abstracts from the specific qualities of a subject matter, and it selects some relations and neglects others'' (pp. 138–139).

Any science, including computer science, succeeds by constructing formal mathematical models of their subject matter that eliminate inessential details. Inasmuch as such details constitute information, albeit irrelevant information, we might call such elimination *information neglect*. It is our contention, however, that computer science is distinguished from mathematics in the use of a kind of abstraction that computer scientists call *information hiding*. The complexity of behaviour of modern computing devices makes the task of programming them impossible without abstraction tools that hide, but do not neglect, details that are essential in a lower-level processing context but inessential in a software design and programming context. The reason they can be essential in one context and inessential in another is because of abstraction and information hiding tools that have evolved over the history of software development.

Abstraction Through Information Hiding

The concepts behind today's notion of a stored program computer date at least from the 19th century with the work of Charles Babbage, who designed an *analytical engine* that would perform complex arithmetic calculations using its *mill* by following instructions encoded in its *store*. These instructions, transmitted to the store through the use of punched cards, amounted to a *program* for the mill. Unfortunately, the design of the analytical engine was beyond the capability of anyone to build at the time, and it remained just that, a design. Today, of course,

Babbage's notions of *store* and *mill* find embodiment in millions of computers as *memory* and *processor*. Had Babbage been able to realize his design, it would not have been an *electronic* realization, but it would have been a stored program computer nonetheless. We would grant it such a status because the notion of a stored program computer itself is an abstraction that can be realized by devices in quite disparate realms. A computer can be realized, for example, by devices consisting of cogs and gears, or by micron-level integrated electronic circuits, or by devices consisting of DNA molecules, etc. The implementation of the device is irrelevant to its functional description as a pair of interacting modules whose combined objective is to carry out computations. This functional description is an abstraction because it hides details about how the computation is modeled and carried out.

A similar point can be made about the *computational processes* that carry out computations. A computational process is an instance of a program execution, and as such comprises events occurring in some medium that can be interpreted as input/output behavior. For a digital computer these are electronic events occurring in a substrate of semiconducting elements. Computational processes can be embodied by other media, for example, mechanical adding machines, slide rules, Babbage's analytical engine, or DNA molecules.

A computational process for an electronic digital computer is described statically, and at varying levels of detail, in textual artifacts written in programming languages. Depending on the type of programming language, whether it be machine language, assembly language, or any of a number of kinds of higher-level languages, the elements of computational processes that are described in textual programs might be as basic as binary digits (bits) and machine registers, or as familiar as telephone books and shopping carts. Whatever the elements of computational processes that are described in textual programs, however, they are never the actual, micron-level electronic events of the executing program; textual programs are always, no matter what their level, *abstractions* of the electronic events that will ultimately occur. Since it is a practical impossibility for a textual program to describe the electronic events, every such program describes a computational process as an abstraction that hides information or details at a lower level.

Programmers need tools to both model and cause the electronic events that will constitute a running program. Primary tools for this are *programming languages*, but in addition, programmers need ways to view the available computing resources (memory, processor, display, mouse, etc.) as though they had exclusive access to them, and they find these tools in *operating systems*. Many application programmers today require additional tools to handle the complexity of programming over a network like the Internet, and tools are available through *network protocols*. Beyond these basic building tools, programmers need abstraction tools to manage design complexity and to make allowances for program modification and reuse. These tools are available through *design patterns*. As we describe next, programming languages, operating systems, network protocols, and design patterns all make essential use of information hiding.

Information Hiding in Programming Languages

From a software point of view, a *bit*, or binary digit, is the interpretation given to a piece of hardware, called a flip-flop, that is always in one and only one of two possible states. It is an abstraction that hides the details of how the flip-flop is built. Machine language is composed entirely of bits. To specify nontrivial computational processes in machine language is a practical impossibility for humans, and so programming languages with higher levels of abstraction are necessary. One such abstraction is the *byte*, or collection of 8 bits, that hides the details of how the bits are arranged so that they can all be addressed together. Various machine data types, like *word*, *integer*, *longword*, *float*, and *double*, are abstractions that hide the details of how bytes are arranged to represent numerical values of various magnitudes and precisions. A *variable* is an abstraction that hides the details of how a symbolic token can serve as a location for changing data values in memory. A *register* is an abstraction of a collection of bits that hides the details of how the bits are intimately related to the processor and memory. A *machine instruction* is an abstraction of a collection of bits that hides the details of how the bits represent processor operations on registers and variables. These abstractions make programming in certain kinds of languages, called assembly languages, possible.

At a higher level of detail, a *subroutine*, *function*, or *procedure* is an abstraction of a segment of memory that hides the details of how the segment represents a piece of a program that is passed certain parameter values and returns a value as a result. A *pointer* is an abstraction that hides the memory address of a piece of data. These abstractions make programming in higher level languages possible. We've mentioned how *ADT*s are abstractions of data and operations. They can be built on top of the data types, procedures, and pointers available in higher level languages.

Sometimes pointers and variables are used by higher level language programmers in such a way that the memory once referred to by a pointer is no longer accessible by the program—it is garbage. A *garbage collector* is an abstraction of a special process that collects garbage and makes it once again available to the original progam, hiding from that program the details of how this is done.

We've also mentioned how *objects* are abstractions of small software computers. They hide the details of how objects can send one another messages to invoke operations, and how they can be set up to inherit attributes and behavior. Programming with them requires a category of higher level languages called *object-oriented* programming languages. Object-oriented programming is the latest in a history of approaches to programming that attempts to make programs reusable.

The practice of literally reusing pieces of program text has a long tradition in programmers consulting published texts or manuals that list programs implementing various algorithms and data structures. Programmers also reuse software when they make use of software libraries, for example, mathematical function libraries or graphics routine libraries. This use of code libraries is an example of *procedural abstraction*, or the ability to execute code through the calling of named procedures that accept explicitly described parameters and return certain guaranteed results. It is an example of abstraction because the details of how the procedure performs its

computation are hidden from the procedure's caller; since the caller only makes use of the procedure for its results, there is no need for it to know the internals.

A procedure makes its required parameters and guaranteed results known through a public *applications programming interface*, or API. The API makes the procedure able to be ''plugged into'' another program in the same way that a hardware component can be plugged into, say, a slot on a motherboard. Using APIs makes a program modular like a motherboard, and it makes procedures reusable.

Besides enhancing reusability, procedural abstraction also supports changeability. Abstracting a procedure's function from the way it performs the function has the effect of *decoupling* the procedure's caller from the procedure. Thus, if the internals of the procedure ever need to change, the procedure's caller is unaffected, provided that the changed procedure still supports the same API as before. Enforcing procedural abstraction by using APIs throughout a system greatly enhances the system's changeability.

Procedures written in machine languages, assembly languages, or higher level object-oriented programming languages all exploit linguistic constructs that are abstractions of aspects of memory and processor hardware required to specify useful computational processes. A remarkable aspect of all of them is that, not only are they used to describe such processes, but, owing to the complex chain of events involved in their translation and execution, they also bring these processes about. To bring them about requires a monolithic, organizing program, called an *operating system*, that makes the system's computational resources available to application programs in a way that can only succeed through information hiding.

Information Hiding in Operating Systems

The terms ''logical'' and ''physical'' appear often in descriptions of operating systems. These terms refer to different points of view regarding a system resource, like memory, processor, disk space, etc. The *physical* view refers to the actual physical state of affairs. The *logical* view is the view of the physical provided to the application programmer. For resources other than the processor and memory, the logical view is reflected in the operating system's API. This interface hides the physical details of the system from the programmer and provides a view and associated language with which to call upon resources in the course of writing application programs.

An operating system's API *virtualizes* the system resources. Virtualization refers to the process involved in converting a physical view to a logical view. It serves two important purposes. First, as described above, it creates a simpler, abstract view for application programmers. Second, it minimizes resource management problems by allowing resources to be shared by several processes without conflict. In fact, one of the most common differences between the physical and logical views is that a resource may be shared by several processes, but logically, the processes are unaware of each other. Each process can have its own logical resource, even though there may only be a single physical resource.

The details of how a physical resource can be shared by many processes are completely hidden from the processes themselves. The sharing of processors, mice

and keyboards, printers, and network connections is accomplished through *time sharing* the resource. The sharing of memory, displays, and disk space is accomplished through *partitioning* the resource. Both kinds of sharing are possible through information hiding.

The virtualization of devices through the physical/logical distinction finds its highest expression in an abstraction called a *virtual machine* (sometimes called an *abstract machine*). Virtual machines are ones whose components and operations are themselves all abstract, in the sense that they exist only as complete specifications of a machine that *could* exist in hardware or firmware, or simulated in software. Today we are seeing not only the virtualization of devices controlled by operating systems, but *virtualization of operating systems themselves* in programs such as VMware (2006) and Xen (2006). This means that one physical computer can simultaneously run multiple operating systems as diverse as Microsoft Windows, Linux, and MAC OSX. It is important to understand that the multiple operating systems do not merely *share* the physical computer by living on separate partitions of the disk; rather, these operating systems are *emulated* through a detailed knowledge of the interfaces these systems present to their users. The result is that programs written for one operating system can run on a computer controlled by a different operating system. This can happen because modern programs are written in such a way that the details of controlling physical devices are hidden from them and entrusted to an operating system, whether actual or virtual.

The value of virtual machines is found not only in operating systems but also in applications. An example is the Java Virtual Machine (JVM, Lindholm and Yellin 1999), a specification for the Java language that is intended to be realized in software on any hardware platform that aspires to take part in the Internet. In this way, a Java program can be written just once, compiled just once into code the JVM understands, and then run on any platform that has a JVM written for it. The specification of the JVM by necessity ignores the details of *how* Java operations are to be interpreted by individual hardware platforms, saying only *what* the operations are supposed to achieve. It is a quintessential example of abstraction through information hiding in computer science.

Computer scientists like to intone a famous quote attributed to Butler Lampson, to wit: ''Every problem in computer science can be solved by adding another level of indirection.'' By understanding information hiding, we can understand what is meant by ''level of indirection'' here. For the difference between ''indirect'' and ''direct'' has to do with the amount of detail that is hidden when one has access to a computational object. For example, if one has indirect access to a newswire story through a news summary received through a cell phone's web browser, one does not have as much information as one would have with direct access to the story. Computer scientists construct levels of indirection through information hiding. Levels of indirection can therefore be understood as levels of abstraction.

Information Hiding in Network Architecture

Today it is essential that application software developers be able to write software that communicates efficiently with other software over a network, in particular, the

Internet. For example, a program might need to make a connection to a database at a remote location, issue a query to the database, receive the results of the query, and close the connection. Many programmers can do this, but very few understand the complexity of the events their programs cause. This is due to the massive amount of information hiding inherent in network architecture.

The Internet makes use of a *multilayered* approach to communication that reduces network component dependence by arranging components in layers accessible according to strict APIs. Simple communication across the Internet involves passing a message down through layers at the message origin and back up through similar layers (*peer* layers) at the message destination. In more complex communication, intermediate communication entities called *routers* are involved, but the use of layered components is the same. Each layer serves a different purpose and uses a different *protocol* for communication. According to Kurose and Ross (2003), ''A **protocol** defines the format and order of messages exchanged between two or more communicating entities, as well as the actions taken on the transmission and/or receipt of a message or other event'' (p. 8). Communicating entities are often a *client*, or process in need of a service, and a *server*, or process providing the service. Protocols generally involve a scheme for encoding and decoding data. However, they also cover dynamic aspects of communication. Many of the Internet application protocols specify command languages that clients can use to direct servers.

Below is a brief description of the layers and their function. Keep in mind that each layer is designed to depend on the functionality of layers below it without having any access to the details of how lower layers get things done.

- *Application Layer*: provides for communication between client and server processes or between peer processes through enforcement of a given application protocol, for example, HyperText Transfer Protocol (HTTP) for transfer of web pages, or File Transfer Protocol (FTP) for transferring files.
- *Transport Layer*: responsible for transmission integrity by breaking up messages into small blocks at the origin and reassembling them in the correct order at the receiving end through a protocol such as Transmission Control Protocol (TCP).
- *Network Layer*: decides how blocks are to be routed from origin to destination through possibly many intermediate machine links using the Internet Protocol (IP).
- *Data Link Layer*: carries out the delivery of information across a single link over a possibly shared medium.
- *Physical Layer*: responsible for encoding bits onto a transmission medium, whether wires, fiber optics, or radio broadcast, in ways that maximize the transmission rate and minimize sensitivity to noise.

When data is being transmitted across the Internet, each layer except the application layer adds headers to the blocks of data that make up a message. These headers are removed by the peer layer on the receive side. Figure 1 shows the addition of headers by a single layer.

The message shown on the left is being sent by a higher layer. The message, as processed by the peers on the send and receive side, is shown in the middle two

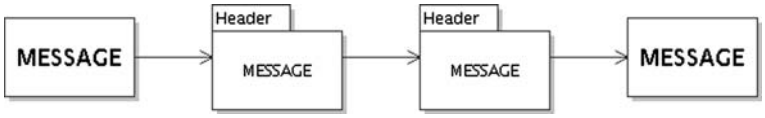

Fig. 1 Communication through a layer

boxes. The original message is recovered on the right, and passed on to the higher layer on the receive side.

The headers, in effect, allow communication between the peers so that they may coordinate to achieve the objectives of their layer. The overall effect is that the two peers on the send and receive side cooperatively implement an abstract service that is used by the layer above. The layer above sends and receives data in a format that is suitable to its own objectives, with the details of how that service is provided hidden.

This layered organization is common in software architectures. It defines an architecture that can be adapted to many different contexts. Each layer can be changed to meet new needs. As long as the abstract interface is maintained, other layers do not need to be changed. In the history of the Internet, there have been many new kinds of applications added, each with software mainly in the application layer. Various technologies have been incorporated into the Internet for faster data transfer through changes to the physical layer. However, due to the independence of layers achieved through information hiding, layers above are unaffected by the changes, other than to enjoy increased performance.

Information Hiding through Design Patterns

Programming languages, operating systems, and networks all exhibit information hiding. They are part of the programmer's development platform and thus in a sense constitute the programmer's toolbox. But information hiding also plays a large role in the way programmers use this toolbox to produce applications. Perhaps this can be seen best by looking at the modern attempt to construct object-oriented systems whose objects are decoupled from each other to the greatest possible extent, promoting changeability and reuse. But in order for objects to be loosely coupled in the first place, the classes to which they belong must be well designed. An object's *class* is an abstraction specified by a programmer in program code. Knowledge about the design of classes has matured enough to the point where best practices have emerged in the form of *design patterns* (Gamma et al. 1995), whose utility lies primarily in their decoupling effects.

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 sort of problem, in which an observed subject (the inventory) 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, called *Observer*, that solves it.

Observer accomplishes its solution elegantly by decoupling the observed from the observing object. In fact, the utility of most of the design patterns in use for object oriented design lies in their 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 exploit information hiding. Objects from classes that are loosely coupled in this way result in reusable and changeable software.

While a design pattern such as Observer is applied at the class level, the notion of a pattern is applicable to larger scale software architecture as well. Architectural patterns have the same objective: increase system flexibility by increasing the independence of the system components. Modern web applications, for example, make extensive use of the *Model-View-Controller* architecture, whereby enterprise data and its structure, called the model, are strictly separated from the views given to users. Model views, in turn, are strictly separated from the modules controlling user interaction. The independence of model, view, and controller components in web applications makes for agile development and change, and is achieved, again, through information hiding.

Conclusions

Mathematics and computer science are similar in that the primary products of both disciplines, and the models that they build, are abstract. There is, however, a significant difference in the type, nature, and use of the abstractions.

The primary products of mathematics are inference structures, whereas the primary products of computer science are interaction patterns. To compare the activity of the mathematician engaged in the manipulation of inference structures and the computer scientist engaged in the manipulation of interaction patterns is to reveal the essence of a distinction between information neglect and information hiding.

The activity of purely mathematical reasoning is characterized by the formalization of inference structures through axiomatization and theorem proving. By contrast, the activity of software development is characterized by modeling interaction patterns in both formal and informal designs, and in various, mostly high level, programming languages. These patterns deal with interactions between elements of software, between elements of software and its users, and between elements of software and a technological foundation that includes hardware, programming languages, operating systems, and networks.

Although mathematicians continue to develop new abstractions and elaborate on the inferences that can be made within existing abstractions, the abstractions

themselves are largely permanent and unchanging. Consequently, most works of mathematics can be described using a single stable and precise formalism whose abstraction objective is characterized by information neglect.

The abstractions of computer science, however, are constantly changing, requiring multiple, multilayered abstractions of interaction patterns, and multiple, less precise formalisms. Information cannot be neglected, but must be hidden in order to deal effectively with constant change and ever increasing software complexity. Change is due not only to the demands of users of software, but also to change in the technological foundations that computer scientists work with, which is, in part, a product of their own efforts.

References

Benacerraf, P. (1965). What numbers could not be. *Philosophical Review, 74*, 47–73.

Carrano, F. M. (2007). *Data abstraction & problem solving with C++*. Boston: Addison-Wesley.

Cohen, M. R., & Nagel, E. (1953). The nature of a logical or mathematical system. In H. Feigl & M. Brodbeck (Eds.), *Readings in the philosophy of science* (pp. 129–147). New York: Appleton-Century-Crofts.

Colburn, T. R., Fetzer, J. H., & Rankin, T. L. (1993). *Program verification: Fundamental issues in computer science*. Dordrecht, The Netherlands: Kluwer Academic Publishers.

Fetzer, J. H. (1988). Program verification: The very idea. *Communications of the ACM, 31*(9), 1048–1063.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Boston: Addison-Wesley.

Hailperin, M., Kaiser, B., & Knight, K. (1999). *Concrete abstractions: An introduction to computer science*. (Pacific Grove, CA: PWS Publishing). Also available online: http://www.gustavus.edu/+max/concrete-abstractions.html.

Hardy, G. H. (1967). *A mathematician's apology*. London: Cambridge University Press.

Kline, M. (1972). *Mathematical thought from ancient to modern times*. New York: Oxford University Press.

Kurose, F. K., & Ross, K. W. (2003). *Computer networking: A top-down approach featuring the internet*. (2nd ed.). Boston: Pearson Education.

Lewis, D. K. (1973). *Counterfactuals*. Cambridge, MA: Harvard University Press.

Lewis, D. K. (1986). *On the plurality of worlds*. Oxford: Basil Blackwell.

Lindholm, T., & Yellin, F. (1999). *The Java virtual machine specification* (2nd ed.). http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html.

Locke, J. (1706). *An essay concerning human understanding*. Illinois: Open Court Publishing (1962 reprint).

Resnik, M. D. (1997). *Mathematics as a science of patterns*. Oxford: Oxford University Press.

Russell, B. (1903). *Principles of mathematics*. Cambridge: Cambridge University Press.

Shapiro, S. (1997) *Philosophy of mathematics: structure and ontology*. New York: Oxford University Press.

VMware (2006). VMware Inc. http://www.vmware.com/.

Whitehead, A. N., & Russell, B. (1910). *Principia mathematica*. Cambridge, England: Cambridge University Press.

The Xen Virtual Machine Monitor (2006). University of Cambridge. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/.