Minds & Machines (2011) 21:241–259
DOI 10.1007/s11023-011-9233-3

Decoupling as a Fundamental Value of Computer Science

Timothy Colburn · Gary Shute

Received: 2 December 2009 / Accepted: 31 December 2010 / Published online: 3 February 2011

Abstract Computer science is an engineering science whose objective is to determine how to best control interactions among computational objects. We argue that it is a fundamental computer science value to design computational objects so that the dependencies required by their interactions do not result in *couplings*, since coupling inhibits change. The nature of knowledge in any science is revealed by how concepts in that science change through paradigm shifts, so we analyze classic paradigm shifts in both natural and computer science in terms of *de*coupling. We show that decoupling pervades computer science both at its core and in the wider context of computing at large, and lies at the very heart of computer science's value system.

Keywords Decoupling · Computer science · Values

Introduction

A primary role of the philosophy of any science is to inquire into the nature of knowledge in that science and how it is obtained. Our inquiry into the nature of computer science knowledge began (Colburn and Shute 2007) by comparing it not to natural science, but to mathematics, with which it shares the distinction of reasoning primarily about abstractions, at least as far as the design and creation of algorithms, data structures, and software are concerned. We observed that while knowledge in mathematics facilitates the modeling of *inference structures*, the primary objective of computer science is the modeling of the patterns that characterize *interaction among objects* in computational processes. This is a crucial

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

difference that shapes the use of formalism and the kind of abstraction used in the two disciplines.

Mathematics employs abstraction to clarify what may and may not be inferred in a given context. To this end, its abstraction objective is to ignore details that are not germane to the inference in question, and to focus on its form alone. Computer science employs abstraction to manage the complex interactions that it studies and creates, so its abstraction objective is to hide, but not ignore, the details of the objects taking part in the interactions. Computer science accomplishes this *information hiding* through its various methods of abstraction—of language, procedures, and data (Colburn 2003).

Notwithstanding its similarity to mathematics in creating and studying formal abstractions, computer science shares with natural science the objective of acquiring knowledge through the application of scientific methods and terminology. As we have shown (Colburn and Shute 2008), the language of computer science is laced with metaphor, including important concepts suggesting direct analogies with natural science. For example, network terminology speaks of information *flow* as though information were a continuous fluid. Programmers speak of program *motion* in terms of jumps, loops, exits, catches, and throws. It is therefore natural to ask whether computer science discovers *laws* governing computational flow and motion in the same sense that natural science does.

We considered this question in Colburn and Shute (2010a). Laws of nature describe phenomena in ways that are general and spatially and temporally unrestricted, allowing natural scientists to explain and predict specific events in time. Computer scientists also need to explain and predict specific events, but these events occur in computational worlds of their own making. The programs, algorithms, data structures, and other objects of computer science are abstractions subject to logical—but not physical—constraints. What computer scientists need are not laws that *describe* the phenomena they bring about; instead they need laws that *prescribe* constraints for their subject matter, so that the concrete processes that ultimately run on physical machines can be controlled. These laws can be described as self-imposed *invariants* that govern computational processes at the algorithmic level.

The program or design invariant that governs an executing computational process reflects a programmer's or designer's *value* choice about what must be preserved in order for the process to be successful. How, for example, does one keep from indexing outside an array, or how do elements in a list retain their order? Progammers and designers impose rules (consciously or unconsciously, often depending on their expertise) on the processes they describe formally through invariants stating conditions that must hold throughout the execution of a process.

In this paper we continue our investigation of the role of values in computer science by focusing on the method of *decoupling*. We argue, in fact, that decoupling lies at the very heart of computer science's value system. All engineering endeavors, governed by best practices, are guided by values. Knowledge in computer science is primarily knowledge of values, and in "The Role of Values in Computer Science", we characterize computer science as an engineering science whose objective is to determine how to best control interactions among computational objects. To begin to understand the role that decoupling plays in controlling computational

interactions, in "Paradigm Shifts in Natural Science" and "Paradigm Shifts in Computer Science" we compare paradigm shifts in natural and computer science, and we see that both involve the decoupling of concepts that were previously linked by conventional wisdom. Natural and computer science differ, however, in what drives this decoupling, and in "Interaction, Dependence, and Coupling" we more precisely characterize what decoupling means for computer science in terms of how it facilitates change. Finally, in "Decoupling in Computer Science" we consider a cross-section of examples of both core computer science practice and computing at large, and we show how decoupling figures prominently in all of them.

The Role of Values in Computer Science

The imposition of values on the subject matter of computer science distinguishes it from natural science and aligns it more with engineering science. This makes sense, because engineering disciplines, like computer science, also create the objects that they study. 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.

While natural science seeks knowledge that helps to best explain observed phenomena, engineering science, including computer science, seeks knowledge that helps to best control interactions. Engineering best practices are statements of value. Just as a programmer engineering a complex computational interaction imposes a value through the maintaining of an invariant, engineering practices as a whole impose value through best practices, which are both learned individually through apprentice experience, and passed from one generation to another in the form of expert knowledge.

Despite the fundamental differences between the kind of knowledge sought by the natural scientist vs. the computer scientist, they can use the same methods to achieve it. As we describe in Colburn and Shute (2010b), acquisition of knowledge about data structures and the algorithms that manipulate them fits the model of Lakatos (1978), in which the "irrefutable" foundation of a scientific research program is given by its *negative heuristic*, while the possibilities for new knowledge, or modifications of claims about things thought to be known, are encompassed by its *positive heuristic*. The nature of knowledge within any discipline reveals itself when new knowledge replaces what was thought to be irrefutable by the negative heuristic, for such shifts in thinking show what sanctions the overthrow of conventional wisdom.

We are interested in characterizing the nature of computer science values. If natural science asks what laws explain natural phenomena, we can ask what sort of values regarding the control of computational processes have been uncovered. In trying to determine what values drive computer science, it is therefore helpful to compare *paradigm shifts* as they occur in natural science and in computer science.

Paradigm Shifts in Natural Science

In natural science, there is universal agreement that what sanctions the adoption of new knowledge is that it better explains observed reality. We give two examples.

Species

Prior to Darwin, biological species were thought of as constant types, or classes, with individual variation within species explained as imperfection. As Mayr (1988) describes it, "In classical taxonomy, species were simply defined as groups of similar individuals that are different from individuals belonging to other species. Thus a species is a group of animals or plants having in common one or several characteristics." (p. 336) Mayr likens this idea of a species to a Platonic universal, or *eidos*, and the Aristotelian language of essence and accident. "The similarity of the members of the species was due to the joint possession of this eidos or essence. Variation was interpreted as due to an imperfect manifestation of the eidos which resulted in 'accidental' attributes." (p. 337)

This concept of a species is an example *par excellence* of a Lakatosian negative heuristic as "irrefutable by methodological decision". However, this should not be viewed as a rationalization for ignoring reality. Sometimes, new knowledge better explains observed reality than established "knowledge," and a *paradigm shift* occurs. When this happens, a previously irrefutable negative heuristic becomes overturned.

Darwin's characterization of species as populations rather than essential types offered a better explanation of observations about species fertility and population stability. As Mayr explains in Mayr (1982), Darwin inferred that "there must be a fierce struggle for existence among the individuals of a population, resulting in the survival of only a part, often only a very small part, of the progeny of each generation." (p. 479) Darwin then combined this conclusion with further facts about individual variability within species and its hereditary nature to posit a process of natural selection, and "Over the generations this process of natural selection will lead to a continuing gradual change of populations, that is, to evolution and to the production of new species." (p. 480)

While the negative heuristic in the pre-Darwinian view of species was a taxonomy of immutable types, Darwin offered a probabilistic model that included natural selection. As Mayr observes in Mayr (2001), "Darwin made a radical break with the typological tradition of essentialism by initiating an entirely new way of thinking. What we find among living populations, he said, are not constant classes (types) but variable populations. Within a population, in contrast to a class, every individual is uniquely different from every other individual." (p. 75)

In the language we shall employ later ("Interaction, Dependence, and Coupling") to describe computer science values, Darwin *decoupled* the concept of species from the concept of typed classes. As part of this new way of thinking, he essentially changed the meaning of terms. In the classical conception, the species *concept* did not have an essence—there was no one characteristic that defined what it meant to be a species—while individual species *did* have essences. Darwin's species concept did have an essence, namely, a species is a population sharing a gene pool, but

individual species themselves no longer had essences. This altering of the concept of species better explained observations surrounding species fertility, variation, and population stability.

Motion Through Space

While Darwin's quest to re-think the concept of species qualifies as a genuine, individual, Lakatosian research program, the sweeping conceptual changes inspired by attempts to explain motion through space occurred gradually over the centuries. Still, the changes can be understood in the language of decoupling just introduced. The Ptolemaic system of astronomy, pioneered by the ancient Greeks, sought to explain apparent variations in the speed and direction of the moon, sun, and planets through epicycles, or the orbiting of these bodies around points that themselves orbit around the earth. Beginning with Copernicus and continuing through Kepler, Galileo, and Newton, epicycles were seen to be unnecessary as earth's perspective was decoupled from general theories of planetary motion. The result was a move from a geocentric to heliocentric and Cartesian coordinate system as well as a set of simple laws (Newton's laws of motion) for modeling reality.

While Newton's laws were impressively successful at predicting planetary motion in an elegant way, even Newton understood that they did not explain gravity. Einstein sought to explain gravity by comparing it to acceleration, and in doing so discovered an analogy with the geometry of surfaces. Comparing inertial to rotating reference frames, Einstein noticed a corresponding difference between Cartesian coordinate systems and curved coordinate systems. However, he believed that the laws of physics should be valid not only from any observer's perspective, but also in any coordinate system, and so his geometric notion of gravity is formulated to not depend on any specific coordinate system. His resulting general theory of relativity has gone beyond Newton's laws in explaining or predicting small anomalies in planetary orbits, the deflection of light by gravity, gravitational redshift, and other phenomena. So while modern astronomy culminating in Newton decoupled earth's perspective from the theory of planetary motion, Einstein carried this further and decoupled specific coordinate systems from the theory of gravitation.

We will show that paradigm shifts in computer science also involve the concept of decoupling.

Paradigm Shifts in Computer Science

While computer science shares with natural science the objective of acquiring knowledge through research programs, it shares with the engineering disciplines the distinction of creating its own subject matter. Rather than studying nature, as in natural science, computer science studies the objects it creates, in the form of machines, algorithms, data structures, and software, to better understand how to control computational processes. So as we describe in Colburn and Shute (2010a), discovery in computer science is not aimed at uncovering new laws of nature and thereby better explaining observed phenomena. Rather, discovery in computer

science tries to identify new values that better allow the control of complex interactions among the abstract objects that participate in computational processes. What sanctions the adoption of new values in computer science, and do these values share an essential characteristic? We consider some examples from the history of computer science: stored program computing, language translation, and structured programming.

Stored Program Computing

A computer can be regarded as a machine that operates on data by following some instructions. Before 1945, a computer was given instructions by wiring its processing elements in specific ways, for example, to decipher encoded messages or calculate trajectory angles for ballistics purposes. If a computer was needed for another purpose, it had to be laboriously rewired.

With the development of the stored program architecture (von Neumann 1945), program instructions became abstractions that resided in computer memory along with the data, and they were executed by loading them into a central processor. Now, using a computer for a different purpose only required writing and loading a new program, not rewiring the computer's hardware. This paradigm shift, which gave us the basic computer architecture that survives to this day, arose because of a desire to make it easier to change a computer's function. Before the shift, changing a computer's function required changing its wiring; after the shift, this dependence was eliminated.

Language Translation

Stored program architecture made computers general-purpose, and so demand for new uses rose commensurately. But new uses required new programs, and the language of computer processors is binary—zeroes and ones. For a human programmer, writing a new program involved the painstaking, tedious, and error-prone process of arranging thousands or millions of zeroes and ones in a format that a processor could accept and execute. The number and complexity of applications demanded by new users of stored program computers threatened to overwhelm the few human programmers who could program them.

With the development of assembly language programs and the ability to automatically translate them into the language of the central processor, programmers could write programs that used English-like symbols rather than only zeroes and ones. For example, to tell the processor to add the integer in register 7 to the integer in location SUM a programmer could write something like:

```
addb r7, SUM
```

as opposed to the equivalent binary instruction:

```
01000000010111000100000101001O
```

While the binary instruction was still required by the machine, a programmer could write the assembly language instruction and have it be translated by a program (called an *assembler*) into binary. This paradigm shift arose because of the difficulty

of coding zeroes and ones. Before the paradigm shift, writing a new program for a computer required knowing and following binary-level instruction formats; after the shift, this dependence was eliminated.

Although assemblers eliminated the need to speak the language of binary, they were dependent on particular machine brand architectures. Thus, an assembly language instruction written for, say, an IBM computer, could look very different from one that performed a similar function for a Control Data computer. To solve this problem, the concept of an assembly language was extended to the first general-purpose and high-level languages such as Fortran and Cobol. These languages eliminated the dependence on machine brand-specific assembly language concepts such as registers and mnemonic instructions like **addb**, in favor of language involving scientific notation (in the case of Fortran) or business concepts (in the case of Cobol). Since these languages were machine brand-neutral, programs written in them could be automatically translated (by programs called *compilers*) to run on any specific machine type. The development of compilers continued the paradigm shift begun by assemblers. Before the shift, writing a program required knowing the assembly language format of a particular machine type; after the shift, this dependence was eliminated.

Structured Programming

The initial introduction of high-level languages freed the programmer from a dependence on machine-specific language details. However, they still required that programmers think about *general* machine architecture while trying to solve a problem in a given application domain. Consider the problem of adding the integers from 1 to 10. The earliest high-level languages required code like this:

```
sum = 0;
i = 1;
top: if (i > 10) goto bottom;
sum = sum + i;
i = i + 1;
goto top;
bottom:
```

Language elements such as the **goto** statement and its associated target label (**top** and **bottom** in this example) impose the requirement that the programmer provide control constructs that need to mirror the layout of the program in memory. The programmer must overlay his or her understanding of the application problem on top of the general concept of a program arrayed in labeled memory elements, with the added requirement of telling the processor where to transfer control at critical junctures. There is thus an inherent mixing of the application problem (adding the integers from 1 to 10) and knowledge of machine architecture in the program. Beyond this limitation, making changes to nontrivial programs with many goto's and labels proved very difficult, as programmers found that modifying a program's control in one place would cause an error to occur somewhere else.

With the advent of structured program *blocks* in high-level languages, programmers were required to take a less active role in explicitly directing the

processor's control. Here is a program that solves the same problem as before using block-structured programming:

```
sum = 0;
i = 1;
while (i <= 10) {
    sum = sum + i;
    i = i + 1;
}
```

Here, the code within the braces {...} makes up a block, in which either all or none of the code is executed, and the **while** construct leaves to the compiler the details of arranging the correct goto's and labels. The development of structured programming (see Dijkstra 1971) was a paradigm shift that allowed programmers to concentrate more on the problem they were solving and less on the details of processor control. Before the shift, programming depended on low-level control constructs that were unrelated to the application problem; after the shift, this dependence was eliminated.

In each of these examples, a paradigm shift occurred in response to a demand for change. The availability of electronic digital computing spurred a demand for new kinds of programs, and the stored program architecture responded to that demand. The new, general-purpose nature of computers resulting from the stored program architecture brought a demand for languages more expressive than the binary language of zeroes and ones, and formal language translation became the norm. The need to maintain and modify complex programs without having to specify low-level control constructs gave us structured programming.

Natural science, too, sees demands for change in its paradigm shifts, but that change is driven by new observations demanding new explanations. Much of what is new in natural science does not challenge prevailing laws or paradigms; new observations are accommodated and accepted into a growing body of knowledge. In computer science, however, many shifts are driven by change per sé. This is not to say that computer science lacks stability or foundation; it is to say that computer science is called upon to provide the tools for modeling a changing world, which, ironically, is often changing precisely because of what computer science enabled through its previous efforts. The challenge for the computer scientist is to adopt values that best accommodate change.

Interaction, Dependence, and Coupling

The primary subject matter of computer science is *computational processes*, or processes that can be specified and studied independently of any physical realization. Processes, whether computational or not, are composed of interactions among objects, so we can also characterize the subject matter of computer science as interaction patterns. The objects participating in these interactions, being computational abstractions just like the processes they carry out, represent not only the entities at the level of machine architecture, like registers, memory locations,

program instructions, numbers, and procedures; they can also represent any object a programmer chooses to model—telephone books, calendars, shopping carts, even humans (consider avatars). Whatever the level, when computational objects interact they collaborate with a common purpose to get something done: look up a name in a phone book, add an appointment to a calendar, complete an online purchase, etc. A primary computer science value is therefore to facilitate interactions among objects to solve problems.

When objects collaborate in an interaction to solve a problem, at least one depends on the services of another. A cell phone interface asks a table data structure for a person's phone number. A meeting scheduling algorithm checks a calendar for event conflicts. A shopping agent requests a debit card balance from a bank. These dependencies are all necessary for the required interaction to occur. Computational object dependencies as such are neither good nor bad from a value point of view. Values come into play when programs and systems must accommodate reality changes.

Modern software systems model reality. While this may seem trivially true today, it wasn't always the case. The early days of computer science saw reality accommodating inflexible software systems. Data input had to conform to rigid columnar standards, for example. Programmers could not divorce the information they modeled in their programs from the data used to represent it—with expensive repercussions for those who did not anticipate the consequences of their date representations leading up to the change to the year 2000. And more generally, programs could not be adapted to any purpose other than those for which they were created.

Change, however, is a fundamental feature of reality, and software systems must accommodate reality changes. Computational objects can accommodate reality changes only by changing themselves, that is, allowing programmers to easily adapt them. But if the computational objects taking part in an interaction are involved in a dependency such that changing one also requires changing another, then accommodating reality changes becomes more difficult. Dependencies among objects which are such that changing one requires changing another are called *couplings*. Since coupling inhibits change, it is a fundamental computer science value to design computational objects so that their dependencies do not result in couplings.

The paradigm shifts in computer science we've mentioned all involve *de*coupling. Stored program architecture decouples programming from physical hardware (wires). Formal language translation decouples programming from abstract hardware (zeroes and ones). Structured programming decouples programming from low-level control (goto's and labels). While these examples are, somewhat narrowly, only about programming, and rather early programming at that, in what follows we will show that the decoupling notion pervades computer science in many of its facets and lies at the very heart of computer science's value system.

As we have seen, decoupling as a response to demand for change is not unique to computer science. In "Paradigm Shifts in Natural Science" we described how natural science responds to the demands of new observations by offering new theories or laws that decouple existing concepts. Computer science, however, models the world rather than studies it. It therefore receives its demand for change

from reality itself when reality changes. It can do this in an agile way only if it designs its objects with minimal coupling. Decoupling the objects that take part in computational processes is therefore a fundamental value of computer science, as we describe next.

Decoupling in Computer Science

We will first show examples of decoupling from some core areas of computer science involving operating systems, data management, software design and development, and software reuse. We will end with some observations concerning decoupling in the wider context of computing at large.

Resource Virtualization

Anything required for the execution of a program is called a resource. The processor, memory, displays, mice, keyboards, disk storage, printers, and networks are all examples of resources, and the primary function of an operating system is their management. The terms "logical" and "physical" appear often in descriptions of operating systems, and refer to different points of view regarding a system resource. The physical view refers to the actual physical state of affairs—it describes the computer hardware. The logical view is the view given to the application programmer, and it is required in order for the physical resource to be shared.

Although a typical computer system has one (possibly multi-core) processor, one addressable space of memory, one keyboard, one printer, etc., many executing processes share these single resources as though each has the resources to itself. For example, several programs (web browser, email client, word processor, etc.) can run concurrently without conflict in the processor or memory, and without contention for the use of the mouse or keyboard. Furthermore, when a programmer writes a program that will run concurrently along with others, he can write it without regard for those others at all, because he has been given logical views of the system resources, and these views have been decoupled from the physical view.

As a result of many years of clever design, operating systems have decoupled processes from each other by creating logical views in which other processes do not appear, a phenomenon called *virtualization*. Virtualization has the advantage of not only creating a simpler and more abstract view of the computer for application programmers, but also minimizing resource management problems by allowing resources to be shared by several processes without conflict.

For example, processes share the same processor through a *time-sharing* mechanism that creates the illusion of concurrency by exploiting the very high speed of the processor along with the fact that many processes spend much of their time in an idle state. Although processes share the same physical memory, they each get a *logical address space* in which the same logical address in different processes gets mapped to different physical addresses, thereby avoiding memory conflicts among processes. As a final example, although several processes share the physical

display device of a computer, each process is given its own logical display window through the computer's window manager, whose *focus* mechanism time-shares the mouse and keyboard input, sending the input to the process whose window currently has the focus. Each of these mechanisms involves a decoupling of a logical view from the physical view of a system.

Data Abstraction and Data Structures

Software developers have long recognized the value of separating data's *use* from its *representation*. For example, suppose you are writing an online shopping application, and the data for the application includes the items that a user intends to purchase. The items can be gathered together in a computational version of a *shopping cart*. In your application, you would like to be able to create new, empty shopping carts, add items to them, remove items from them, and display their contents. From the point of view of the application, how a shopping cart represents data to implement these operations is irrelevant; all the application cares is that the shopping cart does what it is advertised to do. In other words, the application uses shopping carts, but has no need to know how shopping carts are represented.

As the application developer, you might also happen to be the one implementing the shopping carts, so you must choose a representation. Suppose you decide to implement shopping carts using random-access files, and for a while (maybe even a number of years) this approach works fine. But the time comes when your application needs to be integrated with a database system, requiring shopping carts to be stored in database tables rather than random-access files. Shopping cart data objects must therefore be reengineered, but from the point of view of the application program that uses them they will behave just as they did before—they can be created, added to, removed from, and displayed—but the application program has no knowledge of how they interact with database tables. The application program's ignorance of how shopping carts are represented is a good thing, because when the shopping cart representation changes, the application program can stay the same. Separating data's use from its representation, called *data abstraction*, is a classic example of decoupling, and it pervades good program design.

The problem of managing data reveals other, even more fundamental, examples of decoupling in computer science. Programs, being sequences of instructions that execute one at a time on a computer, need data at precise moments in time, namely the moments when instructions that require data are executed. Programs thus expect their data to be available when they need it. At the times they need data, they don't want the added responsibility of creating, organizing, and dispensing the data. The task of using data must therefore be *temporally decoupled* from the task of creating data. This temporal decoupling is the job of a *data structure*, which stores and organizes data so that it can be used efficiently.

Random access data structures (tables, and their big brothers, databases) allow one program component to deal with data production, and another program component to deal with data consumption. A data producing component can store the data in a random access structure, giving the data a key that data consuming components can use to access the data, which they can do according to their own

schedule. The only coordination required between data producers and data consumers is agreement on the association between keys and data. This is non-temporal information so the temporal coupling problem is eliminated.

Other data structures act as data *dispensers* that impose ordering requirements on the processing of data items. A data producing component puts data items into a dispenser in some order and the dispenser hands out the data to a data consuming component in a possibly different order determined by its ordering policy. The need for dispensers often arises when the data consuming component discovers new data items as part of its processing of earlier data items. Then the data producing component and the data consuming component are the same component.

Sometimes a program component requires that the order in which items are placed in a data dispenser be decoupled from the order in which they are removed—a temporal decoupling. A dispenser called a *stack* ("last-in-first-out" dispenser) meets this requirement. A stack gives a component the capability to interrupt what it is currently doing, do something else, then resume where it left off. This kind of processing occurs frequently when nested arithmetic formulas must be processed, so stacks are commonly used in the evaluation of algebraic expressions. Stacks are also indispensable for the very execution of programs written in all modern programming languages. Such programs are composed of calls to numerous subprograms (procedures, functions, or methods), some of which begin their processing but must be suspended while other subprograms are started and completed. Stacks offer a way to keep track of the state of suspended subprograms.

Sometimes data items must be removed from a dispenser and processed in the same order that they are placed in the dispenser, but new data items might need to be placed on the dispenser (though not removed) before processing of removed items is complete. This allows data items to be received but processed at a later time—another temporal decoupling. A dispenser called a *queue* meets this requirement. Queues are used when fairness considerations call for "first-in-first-out" behavior. For example, computer operating systems use queues to schedule the usage of the processor and printer, as well as provide virtualization mechanisms for these resources as described above.

Frameworks

Data structures deal with the temporal coupling that possibly exists between two points in time, namely when the data is produced and when it it processed. Data structures such as stacks and queues are and will continue to be classic objects in computer science. However, modern software design is also faced with a different sort of temporal coupling, not between two points in time, but between an event in time and what processing should occur in response to that event. It is often necessary to decouple these two things, a process that can be called "what/when" decoupling. The need for what/when decoupling arises frequently in the design of frameworks and asynchronous programs, both products of the more recent age of computing.

The labor-intensive nature of software production, and its concomitant expense, combined with the insatiable demand for more innovative and complex software,

has made it imperative that new software projects both make use of existing software and create new software with an eye for its eventual reuse itself. Software developers have long made use of existing software through software libraries of low-level but essential utilities in the form of procedures and functions that do everything from finding the square root of a number, to sorting a list of character strings, to compressing a sound file. A developer who makes use of such libraries writes higher-level code that knows exactly when a library function is needed and what the function will return. This is a classic model of software development, in which the developer writes the high-level application code from scratch and plugs in calls to library functions to make it complete.

The exploding demand for new software has made it evident that many high-level applications share much in common, and that to write all of them from scratch, even while making liberal use of software libraries, is to "re-invent the wheel" over and over. There are countless examples. While there are many manufacturers of microwave ovens with different user interfaces, their software controllers all have similar functionality. Similarly, while different e-commerce shopping websites retail different items with different page presentation, the web application code on which they are based are all doing very nearly the same things. There would obviously be a saving of time, effort, and expense if applications that did similar things in their structure (but looked different on the outside) could share code. Such sharing is the role of a *software framework*.

Frameworks differ from software libraries in that the code that comprises them is "in control." When a software developer builds off of a framework, she only needs to write code that is invoked at the behest of the framework. The code she writes gives the application its distinctive appearance and behavior, but the framework code drives the whole application. There is thus an "inversion of control" when compared to application code that merely makes use of code libraries. But this inversion of control has a consequence: when the framework code is written, the nature of the particular application code is unknown, so the framework must be designed with "slots" and "hooks" that will eventually be filled with code about which nothing can be known. When the overall application runs, the framework decides *when* the specific application code will be invoked, but it must overcome its ignorance of *what* that code does through what/when decoupling. This is a necessary consequence of the framework code being able to work in multiple settings. It takes creativity to design frameworks that perform this what/when decoupling, and much of the work in design patterns (see below) is toward this end.

Asynchronous Programming

Another situation requiring what/when decoupling arises with asynchronous programs. In an asynchronous program, events occur whose timing is independent of the execution of the program. That is, as the program runs, events may occur that interrupt it, but when they occur cannot be known in advance The application programmer knows what to do but has no control over when it is done. This is the opposite of the problem seen by the framework programmer, who knows when something needs to be done but cannot predict what it is. The most familiar example

is programs with a graphical user interface (GUI), which today constitute the majority of desktop programs. The program displaying a GUI has no control over when the program user will click or move the mouse or press or release a keyboard key.

The traditional solution is conceptually simple: after setting up components (buttons, menus, and other controls) in the GUI, the program is driven by an event handling loop. User actions involving the mouse or keyboard are entered into an event queue. The event handling loop just removes events from the queue and handles them by executing the "callback" code provided by the application programmer, who remains blissfully unaware of the existence of the program's event queue and the attending synchronization issues.

Frameworks and GUI programs account for much of the power and richness of modern software, and they are possible only through decoupling.

Object-Oriented Programming

While good software developers have always strived to reduce the coupling among computational objects, the programming languages available to them to achieve this have not always been accommodating. One reason for this has been the failure of languages to include *in their syntax* the ability to explicitly force the hiding of details about one computational object from other collaborating objects with whom it interacts.

Before the 1980s, the programming languages in widespread use allowed the creation of computational objects narrowly defined in terms of the architecture of the machines on which their programs ran. An assembly language programmer could choose from an object "ontology" that included bits, bytes, registers, addresses, and subroutines. A Fortran programmer would add functions, variables, integers, floating-point numbers, and arrays to the ontology. A C or Pascal programmer could talk about all these plus structures (C) or records (Pascal). Although the programming objects available became more abstract and removed from machine architecture details, programmers who used these languages before the 1980s were forced to implement a real-world concept (for example, a bank account) in a language of lower-level concepts (for example, a C structure).

It is always possible for a disciplined programmer to successfully (and painstakingly) model a high-level concept in an ontologically impoverished programming language. But it is likely that the resulting program will be difficult to modify when the real world demands that it be changed, because it will not have been developed in a language that, through *encapsulation*, allows both (1) the convenient representation of high-level concepts and (2) the hiding of their representation details. Such encapsulation is the objective of modern *object-oriented* programming, which enjoyed its first wide-spread use with the language C++, introduced in 1983.

Object-oriented programming (OOP) represented a paradigm shift for programmers. Instead of modeling computational processes primarily through multiple cooperating *procedures* that operate on passive, shared data objects—an idea that overlaid nicely on the idea of machine architecture—OOP models processes

through multiple cooperating active *objects*, each of which has its own state (data) and operations that change that state. In this conception, objects are like virtual computers (rendered in software), with all of the logical malleability afforded to real computers. Just like a real computer can be programmed to act like a word processor in one instance and a 3-D action game in another, an object in OOP can be as simple and passive as an integer or as complex and active as a chat room. A running OOP program can be envisioned as a number of virtual computers sending messages to one another, changing their internal state when required and cooperating to accomplish some overall task.

Good OOP programmers go to considerable lengths to limit what objects will reveal about their internal state to other objects, because the less two objects "know" about each other's internal state, the less they are coupled. OOP makes it straightforward to enforce data privacy through accessibility declarations (`public`, `private`, `protected`) built into the language. While these language elements in themselves afford better decoupling, they are restricted to the internal definition of objects and do not address the broader problem of minimizing the necessary coupling effects of objects when they interact with each other. Some dependency is always going to be necessary when objects interact. Reducing the coupling that accompanies dependency is often the focus of object-oriented software design in the development of *design patterns* (see below). The usefulness of design patterns for reducing object coupling relies heavily on a feature of programming languages called *polymorphism*.

Polymorphism and Interface Types

A programming language supports polymorphism if its operations have different behaviors depending on the types of the objects on which they operate. For example, some languages, when they encounter the expression **a + b**, will perform a mathematical addition if **a** equals **4** and **b** equals **4** (returning **8**), but will perform a string concatenation if **a** equals **black** and **b** = **jack** (returning **blackjack**). The **+** operation is therefore polymorphic.

Object-oriented programming, through its encapsulation of information about objects of similar types, facilitates polymorphism, because the encapsulated information includes not just the state of an object in terms of its *data*, but also the definition of the *operations* that are to be allowed on it. Suppose a class of objects called **Country** is defined, where the operation **compareTo** is intended to compare countries by population, returning **−1, 0**, or **1**, depending on whether a country's population is less than, equal to, or greater than another country's population, respectively. Then the expression **a.compareTo(b)** returns **−1** if **a** equals the country **china** and **b** equals **switzerland**. On the other hand, **a.compareTo(b)** returns **−1** if **a** equals the string **black** and **b** equals **jack** (and an alphabetical comparison is done), and **a.compareTo(b)** returns **0** if **a** equals **4** and **b** equals **4** (and a numerical comparison is done).

Any class of objects for which the **compareTo** operation is appropriately defined can participate in its polymorphic behavior. When the expression **a.compareTo(b)**

is encountered, the *type* of the object **a** is checked to find out which version of the **compareTo** operation should be used. In object-oriented programming, an object's type is its class. In our example, three classes of objects—**Country, String**, and **Integer**—each define their own behavior for the **compareTo** operation.

A class should not be confused with an object; a class definition is an *abstraction* of all that is common to its members, which, as objects, are real computational entities. But object-oriented programming takes abstraction even further than that inherent in the concept of a class. While the classes **Country, String**, and **Integer** seem quite disparate, they do have one thing in common—they all define a **compareTo** operation. Object-oriented programming abstracts this one piece of commonality from these classes and gives it a type name: **Comparable**. Anything that suitably defines a **compareTo** operation is a **Comparable. Comparable**, however, is not a class; it is simply what all classes that define **compareTo** have in common. It is what object-oriented programmers call an *interface* (not to be confused with the term's use in "graphical user interface"). **Country, String**, and **Integer** are all classes that *implement* the **Comparable** interface.

In the expression **a.compareTo(b), a** and **b** are variables. When object-oriented programmers declare their variables with interface types, rather than class types, they do not just maximize their code's polymorphism—they also decouple their code from particular class definitions. If **a** and **b** are declared as type **Comparable**, then **a.compareTo(b)** will make sense no matter what objects **a** and **b** actually refer to. Suppose **a.compareTo(b)** is being used in code to sort a list of objects. Then the same code will work to sort a list of integers as it will to sort a list of country objects. Good object-oriented programmers code with interfaces as much as possible, because the resulting code is both general and flexible. The decoupling effects of interfaces have been exploited and canonized in the development of object-oriented design patterns.

Design Patterns

Design patterns (Gamma et al. 1995) 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.

Consider another example. Users of word processors have multiple ways of accomplishing tasks through the user interface. For example, to cut out some highlighted text, one can use the keyboard to issue a **control-X** key sequence, or one can use the mouse to click on the **Edit** menu and select **Cut**. A good programmer will make sure that the same code for doing the cutting is executed no matter whether the action came from the keyboard or the mouse, effectively decoupling the action's effect from the time of its invocation. This is essentially what is going on in the asynchronous programming example given above in the discussion of what/when decoupling.

But the word processing example has an added twist: if no text has been highlighted, then both the **control-X** key sequence and the **Edit** → **Cut** menu options must be disabled. Similarly, if no text has been cut, then all ways of executing a **Paste** must be disabled. This problem, in which program commands must store state information as well as carry out user actions, also is common enough that an object-oriented design pattern, called *Command*, has been developed to solve it. The solution involves elevating an action to the status of an active object (a command) complete with both operation and state. Doing so completes the decoupling of a command's effect from its source. This allows the command to be easily added to other parts of a user interface, and also allows the command to encompass more state if necessary.

Higher Level Decoupling

The decoupling examples given so far are taken from fundamental areas at computer science's core, involving software design, data management, operating systems, and programming languages. It can also be argued, perhaps somewhat esoterically, that decoupling occurs at computer science's higher levels of focus, particularly involving distributed computing and the continually evolving technological and social environments it offers computer users.

Some of us are old enough to remember when computers were standalone devices that were connected to nothing but an electrical outlet and perhaps a screeching analog modem that communicated with a mainframe computer over telephone wire at a rate of 110 characters per second. The connection of such computers to the "outside world" was woefully weak by today's standards, and so their usefulness was tied (coupled) to what they could do in their physical locations. Today, of course, computers are not only more powerful, but they are portable by virtue of not just their smaller size but their ability to connect to other computers for services no matter where they are. Even for communicating computers that are tethered by various wires to particular locations, the fact that they distribute processing among them makes the question of where the processing occurs both complicated and largely irrelevant. From the point of view of a computer user connected to a network of cooperating machines, information processing has been decoupled from physical location, in much the same way that early cell phone

networks decoupled phone numbers from physical location. (Of course, contemporary cell phone networks *are* computer networks.)

Social interactions began to be decoupled from physical location with the first telephone connection over a century ago. However, such connections only *transmitted* information. With the introduction of computer networks, information could be stored as well as transmitted, and traditional institutions changed due to decoupling effects. For example, desktop (self-) publishing decoupled authors and their readers from traditional publishers; online gaming decoupled competitors from traditional sports teams; blogging decoupled news reporters from traditional wire services; and social networking sites decoupled social interactions not only from physical location (Facebook, Twitter, etc.) but from physical reality (Second Life).

These are all examples of using computers' information storage abilities to decouple traditional personal activities from their traditional contexts. Today, there is anticipatory hype over the coming age of "cloud" computing, in which personal information itself is decoupled from personal location. Cloud computing gets its name from some computer system diagrams that depict system components as tidy box-like icons connected to one another, with one connection, the connection to the Internet, shown as a connection to an amorphous, billowy icon that resembles a cloud. In these depictions the computer system components, including processors, memory, and peripherals, are primary, while the connection to the Internet is secondary. In the vision of cloud computing, the Internet is primary and everything else is secondary. Not only do the various distributed servers making up the cloud take on the information processing tasks of computer users, they also take on their storage tasks, so from the point of view of its users, the cloud decouples information itself from one's physical location. One's information, including documents, email messages, images, movies, etc., do not reside permanently on one given computer. Rather, one's information is ubiquitous, part of a cloud-like ether, while devices able to access it come and go.

With so little left for the typical user's computer to do in the age of cloud computing, it can jettison its older baggage, that is, its typical operating system, and replace it with a powerful web browser, something that Google and its partners are already working on Google (2009). If this vision comes to pass, personal computing, at least, will become decoupled from the principles of computer architecture that have dominated thinking in computer science since the dawn of the modern computing age.

References

Colburn, T. (2003). Methodology of computer science. In L. Floridi (Ed.).

Colburn, T., Shute, G. (2007). Abstraction in computer science. *Minds and Machines: Journal for Artificial Intelligence, Philosophy, and Cognitive Science, 17*(2): 169–184.

Colburn, T., Shute, G. (2008), Metaphor in computer science. *Journal of Applied Logic, 6*(4): 526–533.

Colburn, T., Shute, G. (2010a), Abstraction, law, and freedom in computer science. *Metaphilosophy, 41*(3): 345—364

Colburn, T. & Shute, G. (2010b). Knowledge, truth, and values in computer science. In J. Vallverdu (Ed.), (pp. 119—131).
Dahl, O., et al. (1971). *Structured programming*. New York: Academic Press.
Dijkstra, E. (1971). Notes on structured programming. In O. Dahl (Ed.).
Floridi, L. (Ed.) (2003). *The Blackwell guide to the philosophy of computing and information*. Malden, MA: Blackwell.
Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Boston: Addison-Wesley.
Google Inc. (2009). http://googleblog.blogspot.com/2009/11/releasing-chromium-os-open-source.html.
Lakatos, I. (1978). *The methodology of scientific programs*. Cambridge: Cambridge University Press.
Mayr, E. (1982). *The growth of biological thought*. Cambridge, MA: Harvard University Press.
Mayr, E. (1988). *Toward a new philosophy of biology*. Cambridge, MA: Harvard University Press.
Mayr, E. (2001). *What evolution is*. New York: Basic Books.
Vallverdú, J. (Ed.) (2010). *Thinking machines and the philosophy of computer science: concepts and principles*. Hershey, PA: IGI Global.
von Neumann, J. (1945). First draft of a report on the EDVAC, University of Pennsylvania Moore School of Electrical Engineering.