(1) Throughout your career, you've sought to uncover the fundamental principles of computing. Your book, "Great Principles of Computing," (2015) which you co-authored with Craig H. Martell, attests to this. What led you on this quest? Was it driven by the ubiquitous presence of technology in computing?

Ever since I was a kid in high school I had a strong attraction to understanding the principles of nature and technology around me. I remember giving a presentation to my classmates in high school science club on “basic principles of electronics” – it was so well received that I knew I wanted to be a teacher and my teaching style would emphasize principles. I entered the PhD program at MIT when they were starting work on Multics, an ambitious operating system that would support a computer utility (precursor of today’s Cloud). While working on Multics virtual memory, I was particularly attracted to identifying the key principles on which the new technology rested. I became a spokesman for the need to understand complex operating systems through their principles. That led to me being asked to chair a committee to define an undergraduate core course in operating systems. We accomplished that in 1971 and our report became the blueprint for OS core courses from then down to the present day.

In 1971 I joined the ongoing debate on whether computer science is science, arguing that we needed to emphasize our fundamental principles instead of our technology – just like other fields of science. I set out on a quest to articulate and form community consensus on timeless “cosmic” principles that appear in every generation of computer technology. The ACM Education Board asked me to form a task force to do this job. I took inspiration from the principles-based *Joy of Science* works of Earth Scientist Robert Hazen and Physicist James Trefil. With support from NSF (National Science Foundation) I organized a movement “Rebooting Computing: The Magic and Joy of Computing” that advocated a principles-based approach for helping non computer scientists understand computing. The conversations started in this movement eventually produced a new kind of first course in CS departments, called “CS Principles”, which became part of a modernized AP (Advanced Placement) curriculum in computer science.

Now, to answer your question. No, I was not driven by a desire to explain ubiquitous computer technology. I was driven to see computer science accepted as a genuine science at a time when scientists in many fields were questioning whether computing is science. I’ll discuss this more with one of your later questions.

(2) In your article "The Locality Principle" (2008), you mention, "Locality is a universal behavior of all computational processes: They tend to refer repeatedly to a subset of their resources over extended time intervals." Why is this principle important?

Virtual memory was one of the first new technologies I learned about when I joined the Multics project at MIT in 1965. To me, it was an amazing and elegant technology. Not only did it automate memory management by taking the tedious work of “overlaying” off the programmer’s plate, it provided a foolproof way to prevent memory leaks between programs sharing RAM.  Overlaying was the work of planning moves of pages of data into RAM for processing and back out to DISK when not needed.  It was tedious and error prone work.  I was dismayed to learn that OS engineers were having grave difficulties with the performance of virtual memory – not only did it run slower than programs with manual overlays, early implementations were prone to thrashing, a catastrophic instability where suddenly all programs in RAM were waiting for pages to be retrieved from DISK and system throughput collapsed.  These two problems threatened a beautiful technology with extinction.  I wanted to do something about them.  I devoted the core of my PhD research to figuring out how to make virtual memory outperform the cleverest overlayers and not thrash.

Virtual memory relies on a “replacement algorithm” that decides which page to remove from RAM when a new page must be brought in.  Experimental studies were suggesting that replacement based on use bits of pages performed better.  Beginning in 1966, we started explaining this in terms of a locality idea – that recently used pages are more likely to reused in the immediate future and should not be replaced.  Further experimental studies revealed that locality behavior was found in high-level language programs and in the problem-solving methods programmers used.  In other words, locality was not an artifact of low-level assembly code, it originated in the way the human minds works.  By 1975, there was general agreement that locality is a universal characteristic of programs, as quoted above in your question.

On a cold winter evening just after Christmas 1965, I was pacing my study trying to understand thrashing.  I suddenly had an Aha! Moment and a mind-shift.   Up to that time, virtual memory operating systems used heuristics to divide the RAM into fixed-size chunks for each loaded program.  The heuristics did not work well and allowed thrashing.  The mind-shift was to turn this around: instead of the operating system telling each running program how much space it got, each running program told the operating system how much space it needed.  This was a shift from command-and-control to demand-and-supply.  I used the term “working set” to describe the dynamic memory demand of a running program.  I defined working set as the pages used by a process in a small time window looking backward from the present moment. Working sets were easily measured with usage bits, which were already part of virtual memory. The working set was the demand; the operating system the supply.  The working set policy loaded programs into RAM only when there was enough free space to hold their working sets, and it protected working-set pages from replacement.  Thrashing is impossible under a working set policy.  In my thesis I gave an intuitive argument that, when programs exhibit locality, the working set policy yielded near optimal system throughput.  A few years later, I was able to prove this.  This was the final (satisfying!) answer to the conundrum I took on in my PhD work. Today the working set policy is the reference model for memory management in operating systems.

Over the years, many experiments in many operating systems always showed that programs exhibited locality.  We never saw a program that used its pages randomly.  Locality enabled L1, L2, and L3 caches to significantly speed up CPU access to RAM, and Internet caches to significantly speed up access to web sites. Locality also appears in the design of computer circuits, where all gates fire based on local signals from neighboring gates, enabling great speed of the circuits. Locality appears in the way we humans do multitasking and context switches in our brains. Locality appears in the responses of ChatGPT to prompts – it returns text from a statistical neighborhood of trained texts that are close to the prompt. Algorithm theorists have shown that a procedure can qualify as an algorithm only if the data being accessed is within a bounded distance of the execution point. In short, locality is everywhere that computational processes access data. We cannot compute without it, and with it we can secure the amazing performance of computing systems and networks.

Truly, locality is a universal phenomenon.

(3) In philosophy of computer science, the debate continues on whether computing is a science or engineering discipline. In your article "The Profession of Computing's Paradigm," you argued with Peter A. Freeman (2009) that computing could combine mathematics, engineering, and science. Does the complex nature of computing indicate the existence of something unprecedented?

The debate over whether CS is a science started in the early 1960s when the first departments of computer science were proposed. An unwritten rule in universities is that new departments are formed only when there is a consensus for doing so among existing departments. In those days, electrical engineering departments often objected to computer science because they thought that they were already doing computer science. Science departments such as Physics or Biology objected because they saw the computer as manmade artifact and not a natural occurrence. Math departments objected because they thought that the only substantive part of CS is math, which is their domain. Thus, new CS departments got formed through political compromises such as combining CS and EE departments or placing CS in an engineering school rather than science school.

Through the 1980s, the field was mostly concerned with getting computing technology to work reliably. Over time, the engineering problems were solved and more attention was put on the scientific principles of computing. As computing principles became codified, many computer scientists were annoyed that they were still treated as technology practitioners, not scientists. I was one of them. I devoted a substantial chunk of my career to articulating the fundamental principles of computer science and showing how computer science meets all the criteria expected of a science field. The logic of all this changed some minds, but not enough to quell the debate.

The turning point came in 2001 when Nobel Laureate David Baltimore, a biologist, started telling everyone that biology had become an information science and the future of biology was going to depend on its ability to master computation. Soon other fields joined in, saying they too are information sciences – physics, chemistry, oceanography, and cognitive science, to name a few. This was an amazing shift. In just a few years, information processes moved from being seen as outcasts of science to being natural phenomena.

An important development that set the stage for this transformation was the emergence of computational science in the 1980s, a time when many sciences sought to advance themselves by applying computing technology to their work. Computational scientists found that computational models of natural processes yielded new insights and, in some cases, won Nobel Prizes. They concluded that computation is a third way of doing science, in addition to traditional theory and experiment. Computational science was well evolved and mature by the time David Baltimore spoke those words.

Peter Freeman and I tried to orient this sea change by calling attention to its roots in engineering, mathematics, and science and showing that the “paradigm” of computing is a blend of the paradigms of its three roots.

In answer to your question, I believe that the debate over whether computer science is science has ended. Nowadays I hear hardly a peep on whether computer science is a science. It is widely accepted that computer science studies information processes, natural and artificial.

(4) I understand you have experience with the ACM curriculum for teaching computer science. With the rise of AI and tools like ChatGPT, some speculate that computer science may become so pervasive that a separate career in the field will no longer be necessary ("its success would be its end"). In light of this, how do you believe computing instruction will evolve?

There is an assumption behind what you quote that I do not accept. That is the notion that computing is so general that it encompasses all other fields. Computing is a domain of science, but does not include other fields of science. It is concerned with information processes that can appear in nature or in machines. Take Biology as an example. Biology is concerned about living things, how they function, how they evolve, and how they interact in ecosystems. Biologists do not see information processes as living things, but as aspects of some living things. The concerns of Biology are different from those of computer science and yet there is some overlap. Biologists and computer scientists see each other as peer collaborators.

There is another, somewhat more abstract fallacy: the thinking of computer scientists for problem solving is universal. Computer science thinking does not transfer automatically into other fields. I saw this first hand when I was at RIACS in the 1980s. We teamed up with fluid dynamics to produce better computational methods for evaluating air flows while designing aircraft. Soon my people were complaining that the fluid dynamicists did not accept them as peers, but only as programmers. What was missing was the computer scientists did not have the domain knowledge of fluid dynamics. They were not able to fully participate the ordinary conversations of fluid dynamics.   Fortunately the impasse was solved when the computer scientists learned enough fluid dynamics to participate fully in the team's conversations.

Now that computer science is accepted at the "table of science" it has a bigger educational challenge than in the past when it did not have to manage and honor its relations with so many other fields.   This will affect the evolution of curriculum.

Many universities now recognize the importance of computing by designate it as a school of computing – not part of science or engineering schools, but its own school.  Computing schools are charged with forming relations and partnerships with departments in other schools – the spirit of computational science expanded to many fields.   These schools are pioneering new curricula.

Another aspect of this interdisciplinary shift can be seen with the Advanced Placement curriculum offered in high schools.  Students who take the curriculum and pass the corresponding AP test get a waiver on the CS1 course at the university they attend.  The new AP in computing emphasizes "computing principles" and the participating universities have changed their CS1 courses from "introduction to programming" to "computing principles".

Another shift is being stimulated by the appearance of generative AI – machines like ChatGPT can carry on natural-sounding conversation.  GPT offers tons of intriguing applications in computation for creative artists, poets, and story tellers.  It has also enabled a jump in plagiarism because teachers mostly cannot tell if submitted work was written by the student or by the GPT entity.  Some educators are calling for other educators to redesign their courses to teach subjects as practices and orient exams and homeworks as demonstrations of expertise in the practices.   When this has been done, students who plagiarize are easy to identify.   It is no longer in the student's interest to try to substitute a machine's output for the student's own work.  This is one way generative AI may motivate curriculum changes.

(5) In your book "Computational Thinking" (2015), co-authored with Matti Tedre, the importance of computational thinking in computer science education is emphasized, distinguishing it from other types of thinking, such as mathematical thinking. You argue that CT is not limited to programming concepts but extends beyond that. However, many computer scientists believe that the critical challenges of computing lie in human relationships. Can computational thinking help shed light on this and improve software project management?

Computational thinking has been defined as the mental practices of solving problems by computers.  This formulation has nothing to do with human relationships.   Can marriage counselling be formulated as an information process and troubled marriages transformed by algorithms?  Can software teams be managed by algorithms?  I think not.

Given computing's emphasis on interacting with many other fields, human relationships become important for the success of those interactions. Often interaction skills are labeled as "soft skills" and take second place in a curriculum to "hard skills" such as designing abstractions and proving they work correctly to solve problems. And yet the "soft skills" are harder to master than the "hard skills"!

Human Machine Interaction (HCI) is an old area of computing focused on designing machines for effective interaction with humans. The goal is interactions that go smoothly and do not cause misunderstandings among human users. A considerable amount of psychology figures in. Will an aircraft pilot under great duress in an emergency interpret the displays properly and take actions to resolve the emergency? Or will the stressed pilot misinterpret a display and make a mistake that compounds the emergency? HCI figures strongly in the research on "human machine teaming", a hot area in AI, where the goal is to blend the capabilities of humans and machines on teams that accomplish goals neither could do alone.

In my opinion, many problems of mismatch between humans and machines arise when the designers do not understand the embodied practices of humans interacting with the machines. The area of design of computations would benefit from designers learning more about how humans engage in practices.

(6) You are skilled at using metaphors to explain your ideas. How has creativity impacted your career? (I think of your "The Beginner's Creed")

Metaphors are a well-established tool for education. They can be used to help people see what is invisible to them. But metaphors must be used with caution. If people take the metaphor too seriously then do not learn how to interact effectively with the thing the metaphor has revealed. Take "the brain is a computer" example. It invites us to interpret all human actions as computations taking place in brains. But we humans do so many non-computational things with our brains. If we are hooked on the metaphor, we try to understand and explain the non-computational with algorithms or information process that do not work. Consider a conundrum faced by roboticists: emotions. There is no evidence that emotions are algorithmic. The best roboticists can do is simulate facial expressions and conversational styles that accompany emotions. A simulation is not the real thing.

You mentioned The Beginners Creed. That is not a metaphor. It is a meditation on the moods experienced by a beginner, the person who comes to a new domain of action for the first time and knows little or nothing about it. The only way beginners can move is to find out what are the rules and then try to apply the appropriate rule in the current situation. Beginners frequently experience an array of moods around their inability to perform well, such as anxiety, insecurity, anger, embarrassment, resentment, and resignation. The point of the mediation is to make it obvious that these moods occur and can completely sidetrack you in learning how to become competent in the domain.  Many of my students have told me that they have experienced all those moods in their graduate courses and research projects.  They often tell me, "I wish someone had told me this a long time ago!"

(7) Looking at your professional career, you were never stuck in one specialty; you covered many areas of computing. What would you say to those generalists who aspire to penetrate the ultimate principles of computing?

In the time I entered, computing was not yet accepted as separate field.  It was in the process of being established.  Even the title "computer science" was not the accepted name.  I was in the first generation of graduate students who were educated in this new field.  To be a good computer scientist amid this uncertainty, I determined that I needed to remain familiar with all the segments of the field and what they were up to.  Many of my colleagues expressed the same conviction.  My initial specialty in the field was computer operating systems.  I have maintained my standing as a teacher in operating systems since then.  But I have also kept up with the ideas in play in all the specialties throughout computing.  That has obviously become a challenge as the field has grown and matured.

I've 60 years I've been doing this, I've seen a lot come and go.  I've witnessed many cycles of fashions and concerns.  Try as I might, the field continues to surprise me with new things popping up, new things to learn.  I think of myself as an expert on what has happened in past computing and a beginner in what is appearing.

To everyone, generalist to specialist, I can only recommend: be a beginner every day and enjoy the surprises.

(8) You wrote a book "The Innovator's Way" with Bob Dunham (MIT Press, 2010).   That does not seem to have much to do with computer science.  What's up there?

I have developed a dual career, with expertise in computer science and in innovation leadership. In the early 1990s, I was teaching evening classes at George Mason University to students who had full time jobs during the day.   I discovered that many of them felt like they were in a box with their employers – they were under pressure to generate innovative ideas and yet whenever they had one and presented to the group no one seemed interested.  This made them frustrated about their self-worth and anxious about their careers.  Some of them asked if I could help.  I drew on my studies with Dr Fernando Flores and designed a course for them whose central question was "how do I produce innovations?"  I worked with a new interpretation, that innovation is adoption of new practice in a community.  Innovation is not invention of artifacts that might be sold to the community.   This shifted the question from "how do I come up with a novel invention" to "how do I get my community members to change their practice". I was able to teach students how to do that and they were able to generate innovation adoptions. Many students said that the course not only helped them resolve their innovation conundrums, it also transformed their lives. The students liked the course so much they founded an alumni club we called Sense 21, short for "new common sense for the 21st century". We met monthly for the next ten years to discuss issues on their minds in innovation. Each year a new cohort of course graduates joined the group. We disbanded in 2002 when I left George Mason and went to the Naval Postgraduate School. No one ever asked me to form an alumni group of graduates of any other course I taught. There was something special about the course material and it gave me great joy as a teacher to witness so many students having breakthroughs in how they conducted their lives and work. I kept teaching this at NPS with a new audience and eventually wrote down what I had learned as a book (*The Innovator's Way*). I have continued to teach this and am now working on another book that goes well beyond the first in its coverage of the practices of innovation leaders. So you might say I have mastered both computer science and innovation. And they are not so far apart, are they? After all, computing is involved in so many innovations.