Military Operations Research Society (MORS) Oral History Project: Dr. Peter J. Denning

Captain Wayne Hughes, FS, US Navy (retired)

Naval Postgraduate School, WHughes@nps.edu

Dr. Bob Sheldon, FS

Group W, Inc., bsheldon@groupw.com

MILITARY OPS RESEARCH HERITAGE ARTICLE

INTRODUCTION

Oral histories represent the recollections and opinions of the person interviewed and are not the official position of MORS. Omissions and errors in fact are corrected when possible, but every effort is made to present the interviewee's own words.

Dr. Peter J. Denning is a Distinguished Professor at the Naval Postgraduate School (NPS) in Monterey, California, where he chairs the Computer Science department. He was President of the Association for Computing Machinery (ACM), the leading professional society for computing, from 1980 to 1982 and is a Fellow of ACM. A listing of Dr. Denning's publications and a more complete background is available at http://denninginstitute.com/denning/.

Three previous oral histories have focused on aspects of Dr. Denning's career: "Peter J. Denning Interviewed by David Walden," October–December 2012 Volume 34, Issue 4 issue of the *IEEE Annals of the History of Computing*; ACM Oral History Interview Number 20: Peter J. Denning oral history on August 6, 7, and 8, 2007 conducted by Arthur L. Norberg; and "An Interview with Peter Denning: Building a Culture of Innovation," *Ubiquity*, an ACM publication, April 2004.

This interview was conducted on June 18, 2018 in Dr. Denning's office at NPS in Monterey, California.

FOREWORD

By Wayne Hughes, FS

MORS conducted this interview of Professor Peter Denning at my suggestion. Professor Denning is widely recognized in the computer science community, but he is not as well-known in our operations research (OR) community.

Over the years I have watched how advancements in computers and computer applications have enhanced and enriched OR. This goes all the way back to 1963 when as an OR student I did a field trip to work on the crucial "Cyclops" study chartered by Secretary of Defense Robert McNamara and led by a distinguished analyst, Jim Larkin, who was attached to the Secretary of the Navy's Office of Program Appraisal. FORTRAN was the language de jour and debugging was a huge drag on swift computer-assisted analysis. The Naval Postgraduate School (NPS) had one bulky CDC 1604 computer that filled a huge room in Spanagel Hall. Faculty and students had to queue up to run their code on it night and day. When I came back to the OR department in 1979, there had been vast advances—many computers and computer labs. I had seen some of the advancements in the fleet and OP-96 in the Pentagon, but unlike Peter Denning I had no awareness of what was yet to come. In our campaign analysis course, we have taught ever-more-complex operational and tactical problems as the students' tools advanced. The recent students are challenged beyond anything I gave them in the 1980s because the technology of spreadsheets, PowerPoint, Google maps, combat simulations, and artificial intelligence gives them tools beyond anything imagined by Morse and Kimball when they described OR's successes in World War II, or B. O. Koopman described in the original edition of *Search and Screening*.

Many of these advancements and many of the challenges that accompanied them are described with remarkable clarity by Peter Denning in this interview. When I was Dean of the school in which our Computer Science department resides, I helped to bring Peter and his wife Dorothy to our campus. I knew his reputation and the worldwide respect he had earned, but after this interview I have to confess I didn't know the half of his talents and experience.

MORS ORAL HISTORY

Interview with Dr. Peter Denning; Captain Wayne Hughes, FS, and Dr. Bob Sheldon, FS, interviewers.

Bob Sheldon: First of all, please tell us your parents' names and where you were born and raised.

Peter Denning: My parents were James Denning and Catherine Denning. I was born in Brooklyn in New York City. Our family quickly grew to four children and we outgrew our Queens townhome. We moved to a large house in Darien, Connecticut. My third sister arrived in Connecticut.

Bob Sheldon: Tell us how your parents influenced you.

Peter Denning: I think I was pretty independent, but my father had a strong influence on me academically. He always

celebrated when I came home from school with good report cards. By middle school I was ignored by the other students because I wasn't very good with athletics and was rather nerdy with my growing interests in science and math. Somehow I didn't fit their model of a normal kid. So I kept to myself or hung out with other nerdy types. My mother was very helpful in listening to me and telling me that these other kids didn't know what they were talking about. I had talents that they didn't appreciate, and my parents wanted me to ignore the negative assessments and pursue my talents.

Bob Sheldon: What did your dad do for a living?

Peter Denning: He was a corporate lawyer working for RCA Corporation. He commuted daily to New York City. In 1963 he got a vice president position with Universal Studios and moved the family to Santa Monica in Los Angeles. I stayed behind on the East Coast because I was a junior at Manhattan College and did not want to switch. The rest of the family moved to California.

Bob Sheldon: Did your dad take you on tours of Universal Studios?

Peter Denning: Yes, we went there once or twice. It was interesting to see behind-the-scenes scenes!

Bob Sheldon: Where did you go to grade school, junior high, and high school?

Peter Denning: My grade schools were just normal schools available in my town, Darien, Connecticut. In 1956, when it was my time to go to high school, my parents sent me to Fairfield Prep, an all-boys Jesuit college prep school in Fairfield, Connecticut, about 20 miles east of Darien toward New Haven. Fairfield Prep was much more of a homework-intensive and learning-intensive environment than the middle and high schools in Darien.

Bob Sheldon: Was that a boarding school?

Peter Denning: No. I commuted up there every day on the train along with a whole bunch of other guys.

Bob Sheldon: Did you do your homework on the train?

Peter Denning: No, we talked a lot about meeting girls. You see, there was another, all-girls school a little farther up the line. So we always talked about how we might meet them. According to my sister, who was one of them, the girls talked about meeting some of us boys. But we were too bashful—we never came together to talk!

Bob Sheldon: Did you have some good teachers in high school?

Peter Denning: In high school, the teacher I most remember was my math teacher, Ralph Money. He was a really good teacher, very supportive of his students. He took a liking to me and said, "You seem to know this math." He was right. I loved math even before I got to high school. I went to the library frequently to get math books, especially math puzzle books. Mr. Money said, "I'm going to accelerate you. Work your way through the algebra book as fast as you can. When you get to the end of the book, I'll give you another book which would be for the next algebra course, and we'll work our way through that." I did pretty well on this acceleration and got about a semester ahead of everybody else with his help. But he didn't stop there. He said, "I want you to join the Science Club." He was the advisor of the Science Club. In the Science Club, I met a lot of other students who were interested in science and math. And he said, "Okay guys, every one of you is going to do a project for the science fair, including you, Denning." I said, "I don't have a project." He said, "You seem to like computers. Why don't you build a computer?" I said, "Nobody seems to know much about computers. All I know is what I read about them in the papers. There aren't any books in the library about computers." He said, "All the more opportunity for you to do something new!" Thus he got me to work on a project to build a computer for the science fair, and I did so. My computer was really simple, basically an adding machine. You give it a series of numbers and it added them up and showed the results on a panel of lights. We took that to the 1958 science fair, and it won first prize. I was quite surprised. It was a small piece of electronics with little capability beyond adding a few numbers. But the judges liked it. I think Ralph Money was influential too, because he walked the halls of the science fair talking up the projects of all his students. People came to look. When he saw the judges coming, I think he intensified his campaigning and got crowds to come over. When the judges arrived, there was

already an impressive crowd at my exhibit. He was very good about doing marketing for the projects of his guys.

Bob Sheldon: Did you save that computer for a museum piece?

Peter Denning: No, I didn't save anything. Money was not interested in the past. The moment we finished our post-science-fair celebrations he said, "What are you going to do for next year's science fair?" "Another science fair?" I asked. He said, "Yes, of course. We never stop around here." I said, "Okay, I'll build a more advanced computer." I decided to build a computer that solved linear equations of the form AX=B. You entered the coefficients A and the right-hand side B on dials and pushed the start button. The machine, which was made of relays from an old pinball machine, clicked, sputtered, clanked, and sparked. After a minute or so, it stopped with the answer (the correct value of X) projected on a small tissue-paper window. To do this, I stored X on the pinball machine's scoring relay, a stepping relay that showed a light through a stenciled number on the scoring wheel. It was easy to project the result onto a small screen.

Bob Sheldon: Were you using some Boolean logic?

Peter Denning: At that time Boolean logic was pretty much unknown outside of the circle of logic circuit designers. I wasn't part of that circle. I was just a kid in school, and the only electronic parts I had access to—or I thought I had access to—were cheap pinball machine parts. My family had a neighbor who lived down the street who had a pinball machine in his basement that he wasn't using. I went on a campaign to convince him that he should donate it to me for my science project. I eventually convinced him and got the machine. The machine had a mass of relays, including stepping relays, and a score-relay. The stepping relays were ratcheted wheels that rotated one click with each electrical impulse, thereby recording a number as a wheel position. For example, if I sent the stepping relay three impulses, it clicked three times and recorded the number three. There was a reset input that sent the wheel back to zero. The score-relay was a stepping relay whose wheel contained a stenciled number punched at each position; a light shined through the selected position to display the player's score. I used that relay to display the result computed by my machine. I figured out how to build my machine from those parts. Remember my equation AX=B? When you dialed in the numbers A and B on the control panel of the machine, one stepping relay would record A and another B. I set up a wheel, driven by a clock motor, that in one rotation would subtract the A value from B. When B got to zero, the solution X would be the number of rotations of that wheel. That number was stored on the score-relay so that you could see the value of X. Pretty cool, huh?

Bob Sheldon: We could call you a "pinball wizard," so to speak.

Peter Denning: I doubt that. My machine solved a linear equation. It did not play a pinball game! That machine got the grand prize of the Southern Connecticut science fair in 1959. As a reward, they sent me to the New England Science Fair with that machine. I'm pretty sure that the pinball-parts essence of my machine helped with the prize. As the machine was solving an equation, its relays made all the clicking, rattling, clanking, and thumping of a pinball machine. On top of that, the rotating wheel to find X sparked every time it moved to a new position because those pinball stepping relays used a lot of power. The sparking filled the air with an ozone smell. The machine's noises and smells attracted crowds. I'm sure that influenced the judges. I also had fun with the crowd because occasionally those sparking contacts would melt shut. I had to stop the machine and open the contacts with sandpaper to get it going again. The crowds loved that.

Bob Sheldon: Did you make any more computers after that?

Peter Denning: One of the judges who awarded the grand prize said, "This is really cool. What are you going to do for next year?" That would be my senior year; this was my junior year. I asked if they had any suggestions. The judge said, "Go to the next level with a computer that solves quadratic equations?" So I said, "What the heck. If they say quadratic, I'm going to go one better on them. I'm going to solve cubic equations." So for my senior project I set out to build a computer that would solve cubic equations. I couldn't do it with relays any

more. It would have been too complicated and would require an enormous amount of power. I had to do it with all-electronic circuits. Think of a cubic equation. It has terms for a constant, X, X squared, and X cubed. Each term has a coefficient. The sum of the four terms multiplied by their coefficients is zero when X is a solution of the equation. My idea was to represent each term with a sine-wave signal whose amplitude depended on the coefficient. I represented negative numbers by phase-shifting the sine-wave signal half a wavelength so that a positive would cancel a negative when mixed. I mixed all these signals and blended them into a single voltage that registered on a voltmeter. To solve an equation, you would enter the four coefficients as dials on a panel. Then you would "tune" a knob marked X until the voltmeter read zero. The setting of X producing the zero would be a solution to the equation. Any solution to the cubic equation would, by definition, evaluate to zero. To accomplish all this, I had to design electronic circuits using triode vacuum tubes to generate the signals, phase-shift them, and mix them. Although it felt like a Rube Goldberg contraption, the machine worked pretty well. To help the judges at the science fair, I prepared a list of sample cubic equations that the machine solved well. I took the machine to the 1960 science fair. The judges looked at it, stone-faced without much comment. The next morning, I returned and discovered that my machine received only a second prize. I was flabbergasted, surprised, and disappointed.

Bob Sheldon: Who got first?

Peter Denning: A guy who built a binary counter. I said, "My machine does so much more than a binary counter." I collared one of the judges and told him what my machine did. His eyes opened, and he said, "Oh, I didn't understand what you had done. I wish we had known." Then he said, "I hope you now have learned something about marketing." He was referring to my lack of posters, signs, presentations, and other means to communicate what the machine did and draw crowds to see it in action. I should also note that Ralph Money had left Fairfield for another school and was not at the science fair walking the halls promoting his students' projects. I did not realize until then how important his marketing was to my machines getting prizes. That became my takeaway from that science fair. You could have the world's most advanced computer, and if nobody knows it and nobody appreciates what it might do, it's useless.

Bob Sheldon: With that background, how did you look for a college?

Peter Denning: When it came time to look for a college, I wanted to continue in the Jesuit tradition that I loved while I was in high school. But there weren't any Jesuit engineering colleges. The Christian Brothers ran a good engineering school at Manhattan College in New York City, so that's where I went. My father wished that I had applied to Massachusetts Institute of Technology (MIT), but I said, "No." I wanted to continue my Jesuit type education. I liked it. Manhattan College had a very good engineering curriculum oriented around professional engineering practice. I took electrical engineering because that was the closest to electronics and computers. They had no computer courses. The closest I got was in senior year, when I got to use a newly-donated LGP-30 computer in a project for a teacher. The closest I got in electronics was a transistor course. Computers were being made of transistors at the time. However, our transistor course did not touch on anything computational. It was really how do you design circuits such as amplifiers and radios with transistors instead of vacuum tubes. It wasn't until graduate school that I got into a computer course.

Bob Sheldon: What was your undergraduate degree?

Peter Denning: It was electrical engineering. When I finished up at Manhattan, I applied to MIT. My father welcomed this saying, "Finally you applied to a school that will really challenge you." Most of the faculty at Manhattan tried to discourage me, saying, "No. You shouldn't go to MIT. Previous Manhattan graduates who went to MIT never made it through the PhD qualifying exam which you have stated your interest in." Fortunately for me, the dean said, "I think you can do it." That was what I wanted to hear. I took his advice over the skeptical faculty and went to MIT.

Bob Sheldon: Was it a PhD program when you started out?

Peter Denning: No. The way MIT worked was to put me through the one-year master's program. At the end of the first year I took the PhD qualifying exam. Then I got into the PhD program.

Bob Sheldon: What kinds of courses did you take in that master's program?

Peter Denning: I focused on courses to prepare me for the PhD qualifier exams. The qualifier was based on seven core courses that were in the undergraduate program: besides the electrical engineering topics I had covered at Manhattan, these courses covered material I had not encountered including thermodynamics, antennas, communication theory, and discrete math. I took all those core courses so I could be prepared to take that qualifying exam. Fortunately, the core courses also counted toward a master's degree if you didn't have them already as an undergraduate. The big benefit for me was to learn electrical engineering a second time from an MIT perspective, which contrasted with the pragmatic perspective at Manhattan College. Whereas Manhattan College focused on the design of circuits, MIT focused a lot on key principles: symmetry principles, mathematical principles, electrical signal principles, frequency analyses, Fourier transforms, and conservation laws. Manhattan's was a practice-oriented curriculum and MIT principles-oriented. I had the benefit of both. I felt like I knew electrical engineering a lot better by the end of that first year at MIT.

Bob Sheldon: Did you actually build some circuits and test them out see if the theory matched the measured results?

Peter Denning: We did a little bit of that at MIT. Jack Dennis, a gifted computer systems architect and designer, who was also my advisor, had a lab where he built experimental circuits for dataflow computers. I got to fiddle around a little bit there, but not anything significant. The labs at Manhattan College were more extensive. I was grateful that I had the opportunity to take electrical engineering all over again from a totally different perspective and learn it a lot deeper than I would've if I just had one or the other.

Bob Sheldon: How did you like the Boston area as compared to New York City for a place to be a college student?

Peter Denning: On balance I liked the New York City area better than Boston. Each city has its own personality and idiosyncrasies. In my senior year at Manhattan College, I lived in the dorms because, as you recall, my parents had moved to California. I took a part-time job during my senior year driving school buses for the Riverdale school, which adjoined Manhattan College in the Riverdale section of the Bronx. Every morning at 6:30, I'd drive downtown and pick up seven or eight kids in a station wagon and bring them up to school. Then at 3:30 in afternoon, I picked them up again and took them back downtown. Then I came home and did my studying. I picked up a little money from that. As a result of that, I got to know a lot about New York City, especially about how to drive around in it. I came to understand the characteristic driving habits of New Yorkers and adjusted my own driving habits to be a safe New York driver. When I got to Boston, I had to learn their mindset and driving practices. Some were sharply different from New York. For example, New York cabbies positioned themselves opposite empty spaces in the neighboring lanes so that they could suddenly change lanes without hitting each other. Boston cabbies drove tight patterns. New Yorkers had more respect for pedestrians in crosswalks than Bostonians. I saw more accidents in Boston than in New York. I liked Boston less. When I finished my MIT degree, I was more than happy to get out of Boston. I joined the electrical engineering faculty at Princeton, which was well away from the city and had a much more appealing ambiance.

Bob Sheldon: After the qualifying exam at MIT, then what?

Peter Denning: I came to MIT in 1964, got my master's degree and passed the PhD qualifiers in 1965, and got my PhD in 1968. I was in the PhD program for three years.

Bob Sheldon: How did you choose your dissertation topic?

Peter Denning: I was working with Jack Dennis. He was leading a project in his lab to build a time-sharing system and was part of the architecture team for Multics. He had some fascinating questions about how to design systems and manage their resources to get best possible performance. From my little computer project at Manhattan, I knew that programming

for a rotating-drum computer was hard and tricky because if you weren't clever, rotational latency delays would accumulate and kill performance. For my master's thesis topic, I simulated a rotating-drum memory storage system to learn how to organize files in the memory for fastest retrieval and what policies for scheduling a queue of drum system requests would yield fastest response time. For my PhD subject, I continued in the same direction with a much-expanded horizon: the general resource allocation problem in a multiprogrammed computer system. Jack Dennis and his colleagues were in the middle of designing the Multics project and had numerous performance questions. Multics was an early production-level time-sharing operating system. They didn't know how to do a lot of things. Memory was one of their biggest mysteries. The system allowed multiple programs in memory, executing in parallel, all with different memory demands. How do you organize the allocation of memory and the data transfers with the disks so that you don't create disk bottlenecks? Disk bottlenecks will kill performance on those systems. This question was a natural follow-on after my master's thesis about scheduling data transfers for disks and drums. Now we were looking at the whole problem. How do you schedule everything so that we can make the system run efficiently? So I picked that topic.

Bob Sheldon: Some people say that students nowadays are spoiled because computers run so fast and they have so much memory storage that the students don't have to deal with some of these problems like you dealt with. So they're missing out on learning the mindset of what it takes to allocate scarce computer resources. Can you comment on that from your studies?

Peter Denning: We're getting ahead of ourselves. I'll discuss later the topic of mindsets and thinking. But I basically believe that today we have managed to cultivate among computer scientists an orientation of detached involvement that makes them oblivious to the factors of good design.

Wayne Hughes: I was a student here at NPS from 1962 to 1964 in OR. We had a lot of connections with computer science. We had CDC-1604 number one. Although the faculty dominated its use during the daytime, if a student wanted to get up and learn something on how to program, he'd get on between 2:00 and 4:00 in the morning. I would bring my punch cards and put them in. The first two times I was there, I would have made a bad punch card. So I learned very soon how undisciplined my mind was and how the computers work if you tell them what they need to know, and they won't if you don't.

Peter Denning: The ever-present problem of the bug.

Wayne Hughes: Yes, indeed.

Peter Denning: There's a marvelous quote in Maurice Wilkes' memoir. Maurice was the builder of the first public English computer, the Electronic Delay Storage Automatic Calculator (EDSAC), at the University of Cambridge. He wrote in his memoir that he remembered the day when he was going into his office and he realized as he was walking up the stairs that he was going to be spending most of the rest of his life looking for mistakes in his programs. He came to the conclusion that programming is really hard because it's so easy to make mistakes, and that the engineers will be spending most of their time finding and fixing mistakes. That was the job. A lot of students today have the same problem. They can't get their programs to work. We've got way more advanced tools today than anything Wilkes had. Our students still get frustrated out of their heads when their programs don't work. They blame it on the professors, the compilers, the operating systems, or the network and don't realize that the bugs come from their own lack of understanding or from inattention to detail. Learning to deal with this is part of the job. Programmers make mistakes all the time.

Wayne Hughes: The more power you have, the more complicated they are and there are more opportunities for different kinds of mistakes.

Peter Denning: With the bigger programs you can now write with these more advanced languages, you can make bigger mistakes. And amidst the complexity, the mistakes are subtle, so you can't see them.

Bob Sheldon: When you finished at MIT, were you recruited by Princeton or did you go down there for a visit?

Peter Denning: I interviewed at several universities. I chose Princeton because it was the

most desirable for my family, who didn't want to get too far away from New York City. Princeton is nice. It's out in the country and it's not too far from New York City.

Bob Sheldon: What kind of position were you recruited to at Princeton?

Peter Denning: Assistant professor of electrical engineering.

Bob Sheldon: What kinds of courses did you teach there?

Peter Denning: I started with two courses. One was an operating systems course. I formed an early partnership with Ed Coffman who was also at Princeton. We started teaching the operating systems course, emphasizing principles. At the time there were many principles still in the process of being discovered; as we explored them for the class, we discovered some pretty deep problems involved in building operating systems. Ed and I wrote a book on that called *Operating Systems Theory*, published in 1973. That book supported much of my teaching. The other course I helped with was Jeff Ullman's course on automata theory. I had taught a similar course at MIT and had a draft book manuscript prepared with Jack Dennis; the book, *Machines, Languages, and Computation*, was published in 1978. The book remained in print for about 15 years and was used in a number of universities. However, we left out the important topic of complexity theory. Had we included that, the book might have lasted longer.

Bob Sheldon: Complexity was a fairly new concept back then. How many students were taking your course?

Peter Denning: Each time I taught that course, there were 15 to 20 students.

Bob Sheldon: Did you get along well with your students?

Peter Denning: Yes. We had smart students at Princeton. They had the usual assortment of student hang-ups and issues. Having recently been a student myself, I could relate to all their issues. Their issues were typically finding time to do all the work or struggling to choose a thesis topic. Because they were full-time students, they weren't grappling with other issues, such as how to deal with a part-time job or full-time job in other hours of their day.

Bob Sheldon: How long did you spend at Princeton?

Peter Denning: I was there four years.

Bob Sheldon: What motivated your transition from Princeton?

Peter Denning: Ambition. In 1972, at the end of four years during which I'd been quite productive, several senior faculty said to me, "You should go up for early promotion." But the department chair and the dean and the provost said, "Forget about it." Princeton had a university-wide cap on the total number of tenured faculty and they were pressing up against the cap at that point. Schools and departments had to apply to the provost for permission to bring promotion cases forward. They told my department "You're going to get one new tenured position slot every five years under the current conditions." In my department, over half the faculty were in the electrical engineering side and my computer science side was in the minority. Thus the next quota for promotion was going to the electrical engineering side. Those did not look like good odds for my ambition of tenure, much less early tenure. I figured I'd put myself on the market because, even if I waited to the six-year limit for getting tenure, it was unlikely that the computer science side of my department would have a quota. Not long after that, I was at a conference and got to talking this over on an elevator with Sam Conte from Purdue. And he said, "Why don't you come to Purdue?" He made me an offer right there in the elevator! It was a 50 percent increase in salary plus tenure at associate professor level in their Computer Science department. I thought it over and finally took his offer.

Bob Sheldon: What was it like to move to Purdue from Princeton? Was that a big transition?

Peter Denning: I liked it because it was a dedicated computer science department. In the Princeton Electrical Engineering department, there was an ongoing internal tension over whether the electrical engineering side or the computer science side should get a particular scarce resource. The question of which side would get the next tenure quota was particularly contentious.

Wayne Hughes: If possible, say a word about the trend in establishing computer science departments as a discipline of its own as opposed to a subdiscipline. About when did that happen?

Peter Denning: I wasn't paying a lot of attention to that at the time. The issue of whether computer science should be established as a department was contentious in universities from the late 1950s. I started participating in the debates around 1970. The first department bearing the name "computer science" was established in 1962 at Purdue. Later that year, according to Sam Conte, Stanford established its own computer science department, led by George Forsythe. Both Sam and George were numerical analysts. Numerical analysis was the branch of computer science most respected in mathematics and the sciences. Their computer science departments started with strong contingents of numerical analysts. Compared to other universities, they had a relatively easy time selling their proposals to start computer science departments. Politics were always important in establishing departments.

Sam Conte was involved with the ACM in the late 1960s as part of a committee looking into the issue of a standardized computer science curriculum. That committee issued "Curriculum 1968" with their answer. With ACM endorsement of a standardized curriculum, those seeking to found computer science departments had a blueprint to work with. Sam drew me into these curriculum discussions.

By the time Sam hired me in 1972, Purdue's Computer Science department was very interested in software and computer systems. That was my area of expertise and I had a good reputation. Sam wanted me to help the Purdue Computer Science department get into operating systems, software engineering, and networking. Those specialties were brand new at the time but looked important in the future of computing.

Slowed but not stopped by their internal political wrangling, other universities established computer science departments. There was slow but steady progress. By 1980, I believe there were about 120 PhD-granting computer science departments.

Wayne Hughes: That's the answer I was hoping for.

Peter Denning: Most every computer science department was birthed after a political battle in the university because there is always great reluctance to start a new department. At least one other department is likely to say, "We already do that." For example, the electrical engineering department would say, "Computers are part of electrical engineering. We already do that. There's nothing new there." And the math department would say, "Numerical analysis is part of math. There's nothing new there." Others would say, "It's just a technology department, not an authentic science department; we don't do technology departments." Somehow or other, the people who were advocating the computer science department had to figure out how to make the case in spite of the opposition. It was a tough slog, but they did it.

You might wonder why some of these new departments wound up in the school of science, others in the school of engineering, and still others in the school of business. The ultimate home depended on how they solved their political problems and found the school most friendly to the new department.

Bob Sheldon: Or which one had the money?

Peter Denning: I think it was more who was going to be the most friendly. The new departments could get the money. Funding agencies had research money and industries wanted to help.

Wayne Hughes: This resonates even with NPS. We have a computer science program going and then an information sciences department going. It was in the same school as the Operations Research Department and Special Operations Department, all under the graduate school of Operational and Information Sciences. That was a good fit and I think we've been highly successful in collaborating and complementing each other, in a great part because Peter understands this applications perspective.

Peter Denning: Back when all these arguments were going on about whether or not computer science should get a department in our university, the classical sciences typically argued that computer science wasn't even a science—it's the wrong name for the field. It should be called computer technology or computer engineering. Engineers countered, "Technology is better. It's not really full-fledged engineering." The founders of the field spent a lot of time arguing against this premise—that computer science is not a science. The famous pioneers Alan Perlis, Allen Newell, and Herb

Simon all argued that objection was just wrong. Herb Simon, a Nobel laureate in economics, finally got so annoyed by this whole argument, he wrote a now-classic book called *The Sciences of the Artificial*, where he showed that there are sciences not built around natural phenomena. He said the other sciences were not thinking clearly—all science is built around phenomena, but not necessarily natural phenomena. In the case of computers, scientists were lining up to study all sorts of new phenomena made possible by computers. Economics, Simon's own field, was regarded as a science and yet all its phenomena are manmade.

Wayne Hughes: I've noticed this conflict. Departments tend to want to go deeper and deeper and narrower and narrower. And yet, it's pretty apparent that all of the great breakthroughs in modern science, technology, and engineering are interdisciplinary. I think we do a good job here at NPS in fostering interdisciplinary education, because military disciplines, antisubmarine warfare, and anti-air warfare, strike warfare, global positioning systems, all of these have many different aspects to them. And if you are so focused on traditional electrical engineering or whatever, then you missed that trend that I think is going on right now.

Bob Sheldon: I have a question from my background. In my studies, I learned about both analog and digital computers. In computer science, how has the view of analog and digital computing evolved?

Peter Denning: When I got into the field, many people in computer science believed that analog computing was going obsolete. Electrical engineers were deep into analog computing. They could represent any differential equation with an appropriate set of circuits and use an oscilloscope to view how any particular variable changed. Computer scientists took the view that they could represent any differential equation with finite differences over a grid, and use a computer to evaluate the grid. Because they could deal with much larger differential equations, and faster, they saw no need for analog-circuit simulators. But analog computers did not go obsolete as we had thought. They were used where a digital machine would not work or would be inefficient. For example, a camera depends on light-sensitive photocells and an accelerometer on pressure-sensitive crystals. Today, all the sensors built into our computers (and cars) are analog computational devices that output digital representations of what they have sensed. We like to call them "cyber-physical systems."

Wayne Hughes: When I was a student at the Naval Academy, and that's not too long before where we're talking, everything was analog: the computers aimed the guns; and if they were assisting in navigation, all of them were analog.

Peter Denning: A computer often mentioned by historians is the differential analyzer, built by Vannevar Bush at MIT around 1930. It used gears, levers, shafts, and wheels to solve differential equations by integration. It was very clever—for example, I remember learning from the analog computing guys that it could integrate a function with a ball and a disk. This machine was used by the military to compute ballistic tables for guns, prior to the ENIAC electronic computer. There was a Bush differential analyzer here at NPS. Students worked with it. We have a piece of it in the next room over there as a display.

Bob Sheldon: Let's get back to Purdue. The computer science program was fairly new at Purdue when you got there. Did it grow?

Peter Denning: I arrived there in 1972 when the department was 10 years old. Its founder and head, Sam Conte, was trying to build it out, bringing in new faculty to cover programming languages, operating systems, software engineering, all newly emerging areas of computer science. He wanted to get out ahead of them.

Bob Sheldon: So he did it by recruiting professors like you to bring in various skills.

Peter Denning: Yes.

Bob Sheldon: Was the department growing rapidly while you were there?

Peter Denning: It was growing. I wouldn't say it was rapid. I think we had about 20 people at the time I got there. That grew to around 30 in the next decade.

Bob Sheldon: Did you work on anything outside of computer science while you were at Purdue?

Peter Denning: No, I stuck with computer science and deepened my involvement with ACM. I was first involved with ACM in 1967 as

editor for a group called SICTIME (special interest committee on time-sharing). In 1969, I converted that group to SIGOPS (special interest group on operating systems) and became its chair. In 1970, I was elected first chair of the newly formed SIG (special interest group) board. In 1978, I became the Vice President of ACM and in 1980 the President. ACM occupied more and more of my time as time went on. I still maintained my academic duties of teaching, running research projects, and advising PhD students at a level of productivity above most other faculty members. Nonetheless, colleagues often advised, "Don't do that ACM stuff, man. You'll never get promoted if you do that because it will suck away all your time." I appreciated the advice but managed my time and made it work. I still have a few colleagues today who think I could've done a lot more technically if I'd never done ACM. But I was good at ACM as well as my academics. For example, my ACM editorships turned me into a good writer and editor. I was able to directly influence the way computer science curricula evolved. I led the team that built the ACM digital library, a singular accomplishment among professional societies. I like the way my life worked out.

Bob Sheldon: When did you start getting involved in queueing network studies?

Peter Denning: At MIT. Both my master's thesis on disk optimization and my PhD thesis on working set memory management and the general resource allocation problem used queueing models extensively. I used insights from queueing theory such as Little's Law to develop my working set and locality theories, which became central in operating systems. I found queueing networks especially intriguing because they were oriented toward the general resource allocation problem I studied in my thesis. In the early 1970s, there were some intriguing studies showing that queueing networks accurately modeled computer systems. Ed Coffman and I wrote about them in our 1973 book *Operating Systems Theory*. While doing my master's and PhD theses, I was fascinated watching my colleague Allan Scherr discover remarkable agreement between a machine repairman model and the MIT compatible time-sharing system (CTSS). His finding was astonishing because the machine repairman model left out 99.99 percent of the details of CTSS, and yet it predicted throughput and response time almost spot on. I remember around 1970 Forest Baskett at the University of Texas did a whole thesis with a slightly more complicated network that modeled a small computer system with about three or four servers. He lamented that he could not do larger, more realistic models because they were computationally too expensive. I wanted to couple my working set memory management theory with queueing networks. Could a queueing network model confirm that working set management avoided thrashing and gave near optimal system throughput?

Wayne Hughes: These models got to the heart of the basics, the most important part of queueing models.

Peter Denning: Now we know that the performance is largely governed by the bottlenecks of the system. Scherr and Baskett modeled the bottlenecks, which, in time-sharing systems, were the discs of the memory system. They also modeled the CPU, which rendered service to get the jobs done. And finally, they modeled the users, who submitted jobs to the system. The models were closed systems because all the work came from a fixed community of users. For a performance prediction model, this was all we needed.

Wayne Hughes: I have a computer queueing theory example. When I went on my experience tour in 1963, they sent me to Washington to work on what was supposed to be, probably was, the most important study going in the Navy. And they had a very short time to do it, like six weeks. I got there in the second or third week. They had just hired a guy who was in the Secretary of the Navy's office who was very applications oriented. He came in just as I arrived and said, "Where do we stand?" There was a queueing theory model that was supposed to simulate the Battle of the Atlantic between Soviet submarines that would attack one target, and then another target, and then another target, in queueing fashion. A fellow named Frank Houck at the Center for Naval Analyses (CNA) had designed it for an earlier study and he brought that in. We were doing the computations with a mechanical computing

machine. The director said, "Can't we get this onto a program?" So CNA sent an expert on FORTRAN over to program it, and she said, "Yes, I will have this up in three or four days. I'll be giving you results in a week." This was good news for me because I'd been doing all the hand calculations until midnight or one in the morning. I thought, "Finally I can learn something more than just turning the crank." The director said, "Wayne, keep chugging until she gets this thing debugged." When I left three or four weeks later, she still hadn't got it debugged, and I was still the key man doing the computations.

Peter Denning: Going back to the queueing network, Scherr's insight that the machine repairman would work for the MIT CTSS time-sharing system was based on his desire to model with the simplest possible network you could think of. A quick look at the system configuration would suggest that he would need a model consisting of the users, the CPU, and the disk. That would be two servers plus users, and would have been much more complex computationally. But the CTSS was designed in a way that reduced the CPU-disk combination to a single server. The memory was single-user. When a job got scheduled for CPU, the system swapped it into memory and ran it to completion while loaded. CTSS also set the time slice length long enough that the probability of a queue at the disk was near zero. This is why Scherr got such good results with the machine repairman model.

Scherr graduated from MIT in 1965 and Baskett graduated from the University of Texas in 1970. The tempo of queueing network contributions by computer scientists picked up rapidly after that. In 1975 Baskett, Chandy, Muntz, and Palacios (BCMP) published a now-classic paper in the *Journal of ACM* about queueing networks that set off a new round of algorithm development.

Bob Sheldon: Was BCMP an open Jackson queueing network model?

Peter Denning: No. It was a closed Gordon-Newell network queueing model. The Jackson model was easier to deal with, because it was just a series of open systems strung together in tandem making use of the fact that the output of an exponential server is Poisson. The Gordon-Newell model was closed. The state space was much more complicated. But the theoretical result was magical: the steady-state probability of any system state is a product of terms, one for each server, each using only parameters of that server, and each raised to the power of the number of jobs queued at the server. That was quite an important result.

Unfortunately, computing the state probabilities appeared to be computationally infeasible because of the astronomical number of states in a system. In 1976, Jeff Buzen reported in his PhD thesis that he had found an algorithm that would compute those probabilities very rapidly. He had found a deep structure in the state space that allowed him to calculate throughput, response time, and mean queue length of any server in time proportional to M and N, where M is the number of servers and N the number of users. Everybody had thought it was exponential up to that point. The product-form came with an elegant computational algorithm.

Wayne Hughes: So M times N was actually good news.

Peter Denning: Yes. That's quadratic. With Buzen's algorithm, you could compute performance metrics for realistic size computer systems (for example, 10 servers and 100 users) on a laptop. In fact, I think the first uses of it were done on HP-35 calculators. His friends in the industrial engineering schools modeled factories as queueing networks and could use his algorithm on their portable programmable calculators. This was immensely valuable to consultants. All they needed were the values of parameters and they could then quickly compute throughput and response time curves as a function of the number of users.

Buzen's algorithm came to be called "convolution algorithm" because of its mathematical form. Unfortunately, numerical analysts began to find cases where the algorithm could become computationally unstable due to roundoff error in the machine and give invalid answers. In 1980 Martin Reiser and Steve Lavenberg discovered an alternative algorithm, mean value analysis, that had the same running time as the convolution, but was numerically stable.

Quite a series of breakthroughs, eh? They were all done by computer scientists who were basically looking for algorithms and then trying

to figure out how to compare the model against the real system.

Bob Sheldon: How did you get teamed up with Buzen?

Peter Denning: We met at a meeting at Harvard around 1977, when he was presenting his algorithm at a conference. I thought his algorithm was one of the most amazingly cool things I had ever seen. I wanted to know, "How did you find that? Now that you say it, it's obvious. But it wasn't obvious before. How did you find it?" He said he just kept playing around with it, and one day he noticed a pattern, which was the one he needed. He's a smart cookie.

Wayne Hughes: The genius is to recognize the discovery when you make it.

Peter Denning: Right. Buzen gave us a major breakthrough for computational use of queueing network models. All of a sudden, the math became accessible to the practicing engineer. You didn't need a supercomputer to do it. You could do it on an HP calculator.

Bob Sheldon: Some people call that the network counterpart to Little's Law applied to a simple queueing network system.

Peter Denning: That is a conversation Buzen and I got into. He got into it before I did. He had been discovering similar things to things I had noted. I liked working examples with simple networks because I wanted to teach my students to use the product-form solution. We would parameterize and solve the equation. In these examples, I had noticed that Little's Law, which says that the mean queue length is the product of mean response time and throughput, always seemed to hold, all the time. And utilization law, which says the utilization of a server is the product of the mean service time and throughput, seemed always true. Up to that point I thought of Little's formula and the utilization formula as limit theorems for systems in steady state. Buzen had been wondering if these formulas are laws when used in real networks—invariant relations within the data. Little's paper about the law was pretty mathematical and seemed overkill for practical cases. I had a very pragmatic wine cellar example. I said, "I'm a wine collector and I like aging my wine. The optimal aging time for best flavor is 10 years. I drink a bottle of wine a day and I want to age my wine for 10 years. How many bottles capacity do I need in my wine cellar?" Everybody says it's obvious: 10 years times 365 days equals 3,650 bottles in my basement. That's Little's Law. I advised students that if they were having trouble figuring out what Little is saying, just go back to this example and it becomes obvious why this law holds.

Wayne Hughes: Many of your examples have been, apart from the wine example, engineering and business. Bob asked you earlier if there was an OR connection and there is. As you well know, the OR major includes lots and lots of computer science systems now. One of the great breakthroughs in doing campaign analysis and other forms of analysis is the ability to use Excel and other kinds of spreadsheets. When I first started teaching, the students had to do everything; you couldn't do a differential equation. As soon as you got a spreadsheet capability, you could do difference equations and approximate a differential equation. And the power to do more complicated simulations just grew by leaps and bounds. And what I'm hearing is you're echoing this progress that enabled all kinds of professions to advance as computing systems evolved.

Peter Denning: That's right. Thinking computationally about queueing networks applied to real data led not only to fast algorithms but also to some amazing simplifications. Buzen was asking why this was so. Why are Little's formula and the utilization formula obvious in a network? Others were asking the same question, especially our colleague Dick Muntz at the University of California, Los Angeles. Buzen noticed some other anomalies. All the math of queueing networks invoked the theory of Markov Chain systems to do the solutions—and yet the real computer systems that we applied the fast algorithms to weren't Markovian—they weren't in steady state, weren't ergodic, and the service distributions weren't exponential. Real systems deviated significantly from these basic modeling assumptions. Except possibly for arrivals from the user community, you were unlikely to measure an exponential service time distribution anywhere in the system. Typical servers were distinctly not exponential. Steady state was laughable because demands for computing resources varied significantly by time of day. There is no steady state. Ergodicity was

another one that they all challenged. You could easily exhibit time series analyses that differed from ensemble analyses. Buzen was astounded with all this: "Look at this. Queueing math is so elegant. But the assumptions that make the math work don't apply to real computer systems, and often the discrepancies between assumptions and reality are large. But yet models give excellent results, usually getting throughput to within 5 percent of the real system and response time to within 25 percent. How can this be?" Buzen and I entered into a partnership to investigate this question. In 1976 he wrote a paper about "operational laws," in which he showed that all the queueing theory steady-state limit formulas were algebraically true with real data. He said that the way we collect the data guarantees that those laws are true.

Let me give you an example. To take a measurement of the CPU, we define an observation period of length T. We can then use a stopwatch to record the total CPU busy time B during T. With counters, we can count the number of arrivals A and completions C. In terms of these measurements, we define the CPU utilization as $U = B/T$, the throughput as $X = C/T$, and the mean service time as $S = B/C$. Then the utilization law, $U = SX$, is an algebraic identity. Buzen and I did this for other laws including Little's law, a system response time law, a memory space-time law, and a forced-flow law. These relations are true for all networks. You don't need to invoke Markov Chains. You don't need to invoke steady state. Buzen called these laws operational because every quantity is measurable—you just work with numbers you measure operationally in the system.

That was the first foray into questioning the assumptions of the Markov theory when applied to computer systems, because computer systems seem to violate basic assumptions of the theory. Buzen said, "Maybe there is a whole new set of assumptions. If we knew what they were, we'd find that models using them more closely agree with the real systems, and that would allow us therefore to be comfortable with the validity of the model." He and I began to look carefully at which modeling assumptions were "operational"—that is, directly observable by an appropriate experiment. We wanted to find modeling assumptions that did not rely on unobservable conditions such as steady state or assumptions that could not be tested such as ergodicity. In the end, we found operational assumptions for computer systems that gave the same mathematical equations as Markov Chain theory. In the case of the queueing network, we demonstrated that the three operational assumptions of flow balance, one-step behavior, and homogeneity give us an operational theory for queueing networks. The operational model gave the same product-form solution and the same computational algorithms.

Bob Sheldon: Say more about those three operational assumptions.

Peter Denning: The flow balance assumption says that the number of entries to every system state is the same as the number of exits. That will be true if the observation period begins and ends with the same system state, which by the way is a principle used in regenerative simulations. Regenerative simulations remove the need for correcting for end effects. Flow balance is a pretty weak assumption because it is approximately true for most observation periods. It is the operational counterpart of the Markovian steady-state assumption.

The second assumption, one-step behavior, means that the only state changes you observe are caused by singleton job moves from one server to another in the network. This is also a pretty weak assumption and is used in Markovian theory as well.

The third assumption is homogeneity. It says that if you take a device offline and subject it in isolation to a constant load, the flow you see through the device is the same as you would see under the same average load online. This is intuitively appealing but often does not work. The flow through a device online can depend on how other devices are sending it load. It is quite possible that the online and offline behaviors are not the same and maybe not even close. As you might suspect, this assumption generated a lot of controversy. Critics said it was a hidden exponential assumption, which transformed our operational model back into a Markovian model, contrary to our purpose. We investigated that and found many examples of networks that were provably not exponential, but which satisfied the three operational

assumptions and therefore had the same mathematical, product-form solution. Operational analysis was indeed a new kind of model. Buzen and I were confident that this explained why product-form models worked with real systems—it's because the operational assumptions hold approximately with real systems. We concluded that the homogeneity assumption is not a bad assumption for most systems. And because it is operational, we can measure the error that arises from using the assumption.

Wayne Hughes: Talking about the homogeneity assumption, we apply for modern combat a simple mathematical formula, sort of a descendent of the Lanchester form, and one of the assumptions is the ships on each side are all identical. We've gone so far as to look at different combinations of ships on each side in more detail, like aircraft carriers, destroyers, smaller destroyers, flagships, to look at targeting and probabilities of hit. It was not easy to write, but a student wrote the equations in a matrix form. But the problem was you couldn't get the input data. I'm wondering, in other fields this notion of homogeneity, is it partly because input data is hard to get, so you assume homogeneous circumstances?

Peter Denning: Well, homogeneity is clearly an easy modeling assumption.

Wayne Hughes: And it often gives you vital information.

Peter Denning: A lot of times, at least in computer systems, you could do the offline analysis without even simulating anything. Consider, for example, a rotating disk. When a job arrives, it enters the queue. The service time is calculable from the disk parameters—the rotational positioning time to get to the start of the requested data block, then the transmission time of the block. You can work out the disk service rate for any number in the queue in an offline configuration. You then put that service rate function into the model. It works well because the hardware physics of the disk dominate.

Wayne Hughes: The analog of that in combat is if you solve the equations for homogeneity, you gain all kinds of information. And then you don't try and overload the model, you just do some side analysis.

Peter Denning: That's what we did. Homogeneity is the key assumption set that makes the operationally defined product-form agree with real systems.

Bob Sheldon: When I was a grad student at Cornell and first read the queueing paper by you and Buzen, I didn't know that both you and Buzen were computer scientists. I thought you were industrial engineers or operations research people because this was serious queueing theory, and queueing theory is typically taught in OR departments. Is it common for computer scientists to study queueing theory?

Peter Denning: For my part of computer science, all the time. We build computer systems. One of the big questions of any computer system is, "Does it perform well?" That question applies to supercomputers, laptops, and everything in between. Our theorists approach performance as an algorithmic complexity problem. They will say that the running time of an algorithm is N-squared, N-cubed, exponential, etc. There is nothing in complexity theory to deal with the issue of competition between the different users of a system that queue up for a server. You need queueing theory to answer the performance question for multi-user systems. Queueing theory gives a theory that we know is pragmatic, computational, and useful, and actually works for predicting the performance of the computer system and the network. That's why we got into it. Let's say we are going to build a new operating system. Everybody wants to make sure the new operating system is going to perform well. They will give concrete statements of what good performance means, for example, the response time is 20 seconds or less to anybody making a command. We need to figure out how to design the system to achieve this. What servers are needed? How much capacity in each? Where are the bottlenecks? What will the response time be when 100 users are logged in?

Wayne Hughes: You said something that I've never forgotten. It was you and John Arquilla together. The issue was artificial intelligence and to what extent our thinking machines were going to take off. For example, thinking machines beating human beings and thinking machines playing Go. You had a remarkable comment. You and John together. The comment I remember is, "Yes, machines sometimes can beat human beings, but a machine in

combination with a human being will beat either the machine or the human being." I think that's very important for the future.

Bob Sheldon: After reading your paper at Cornell in 1985, I didn't see your name again until a few years ago when I was working on this problem for the Marines that involved queueing theory. I looked up your name and I found your paper online and saw your vitae here at NPS. Among your teaching interests, you list performance monitoring and evaluation. Is this what you're referring to there?

Peter Denning: Yes. I'm a pragmatist with computing, and computing has to be useful before people will use it. If I come up with a fantastic algorithm or a fantastic supercomputer, or both, and it takes too long to get the answer, nobody's going to use it. Even if it's free.

Wayne Hughes: I think this is the time to remind you of something you said when the Undersecretary of the Navy was here, and you had just celebrated your 50th anniversary in computing. She said, "Can you summarize in a few words your career as a computer scientist?" It may be an oversimplification, but I've never forgotten it. You said, "For the first 40 years, we were developing the systems and the computer capabilities. And for the last 10 years, it's been all applications. And the applications have come to dominate our thinking."

Peter Denning: Yes, in the early days, we were spending most of our effort trying to get the darn things to work.

Wayne Hughes: And now you've got great power and much more reliability, and you spend less time on the development stages and more on applications. Of course, there was no sharp one moment when this happened, but I thought that was a very important comment.

Bob Sheldon: Let's get back on track at Purdue. How long did you stay at Purdue?

Peter Denning: I went to Purdue in 1972 and left in 1983, when I went to NASA Ames Research Center. NASA Ames was establishing a new group called RIACS, acronym for Research Institute for Advanced Computer Science. They were looking for a director; I applied. My wife Dorothy and I had both become tired of Lafayette, Indiana, and enamored of the West Coast. I got an offer from them to be the director. Dorothy had friends at SRI; she applied there and got a job. We were very happy to have so quickly landed two jobs in the Bay Area, and we moved out here.

Bob Sheldon: Were you now more of a manager than a researcher yourself?

Peter Denning: I did both. I spent a lot of time as manager but maintained a research program and maintained my hand as a writer. I was invited to write a "computing science" column for *American Scientist* magazine, which I did from 1984 to 1993. The purpose of the column was to examine computing issues that arise frequently in the context of science.

It's interesting that one of the reasons that I became interested in leaving Purdue was the management responsibilities I acquired when in 1979 I became head of the department. The difference between a head and a chair is the head can make hiring offers over his own signature as long as the salary was available in the budget. A chair can only make a hiring recommendation to the dean, who has to make a recommendation to the provost, who makes the offer. Chairs have a lot of responsibility, but no authority. Wayne can testify to that. The deans have a little bit of authority, but not a lot.

Wayne Hughes: And a lot of it goes right up to the upper administration and even to the Pentagon in Washington.

Peter Denning: When I became department head, I found myself thrown into a completely different world of conversations than anything I had experienced before. This became obvious when the provost had occasional cocktail parties for all the department heads, to help us get to know each other. At these gatherings, other department heads would ask, "What do you guys in computer science work on?" I answered, "We do operating systems, programming languages, databases, networks, and a little computer security." No one was interested in any of these topics except perhaps computer security because they worried about loss of data on their computers. I just could not engage them in a conversation about anything that we did in computer science. They all wound up gathered around the physics department head. He had all sorts of wonderful stories from physics such as black holes and galaxy collisions. As I listened to the physics

chair, my impression was that I didn't know how to talk about computer science in a way that's accessible to non-computer scientists. So I taught myself how to write for *Scientific American* and got a couple articles published there. I took these stories to the department head gatherings … and I still couldn't get anybody involved in a conversation. Finally one day I said to myself, "Maybe the reason I can't get the other chairs engaged is that I'm not working on anything that interests them." I began to wonder if I might do better in a different environment. NASA Ames appeared as a new possibility. It is like a Disneyland of wind tunnels, robots, advanced science experiments, and space exploration. I thought that by getting into the NASA "soup," I'd learn how to associate myself with problems that people are actually interested in. That became a big driver for me, to transfer myself into a different environment, to be stirred into a different "soup," where new conversations would instill me with new concerns, new stories about what's important, taking me out of my academic cocoon.

Bob Sheldon: Did you change the direction of NASA Ames and their focus?

Peter Denning: As we discussed earlier, I went to NASA Ames in 1983 to be the founding director of RIACS. The mission of RIACS was to bring computer scientists to be members of NASA teams working on computing issues of strategic importance to NASA. We became involved early in computational sciences, a term that refers to the computational branch of a science, for example, bioinformatics, computational chemistry, or computational fluid dynamics. RIACS was one of the first centers in computational science. We helped shape the NASA Ames approach in those areas. We also helped on a lesser issue. When we came, my team set up a Unix network with nodes around the NASA Ames site. NASA people learned a lot about networking from the experts on my team. They developed a world-class networking group of their own to support high-performance computing. That was a bit of infrastructure shaping, I suppose.

I really resonated with the idea of computational science, which is that computing was a new way of doing science alongside the traditions of theory and experiment. The use of computation throughout science to make scientific discoveries was a big idea.

In pursuit of this idea, we organized our RIACS researchers into three groups: networking, autonomous systems, and computational fluid dynamics. The networking group was involved in projects that supported the NASA idea of "telescience"—being able to perform experiments remotely with robots and other remote sensing systems. Sending autonomous vehicles to explore Mars is an example. What kind of networking do you need to talk to the robot? How do you conduct scientific experiments at a distance through robots? What if it takes a long time to send messages through deep space, such as 20 minutes one-way to Mars?

Our second group was autonomous systems. It focused on technologies for artificial intelligence including Bayesian learning, sparse distributed memory (a model of human memory), and neural networks for performing human actions such as landing an airplane.

Our third group was the computation group. Its main focus was numerical aerodynamic simulation, the use of a supercomputer to compute airflows around aircraft using the mathematics of computational fluid dynamics. This was very successful for NASA and it paved the way for the Boeing company to design the 777 aircraft without wind tunnel testing. The group also worked with computational chemists designing heat shield materials for the Jupiter probe, which was then able to descend farther into the Jupiter atmosphere than anyone expected. They used the mathematics of the Schrödinger equation to evaluate the bonding strengths of various materials.

Wayne Hughes: Wind tunnel simulation, huh? Instead of a physical system.

Peter Denning: Instead of its technical name "numerical aerodynamic simulation" I used to call it "flying an airplane inside the computer." That phraseology caught people's attention, and they would ask how it works. Nobody asked questions when I said we were working on numerical aerodynamic simulation. It's the same thing as at Purdue, but the more picturesque way of talking got a conversation going.

Bob Sheldon: You learned your salesmanship from that high school science fair?

Peter Denning: Yes, in the sense that I learned from that high school experience that a new computer might not sell if we didn't pay attention to saying what useful jobs it does. It's also the same issue I encountered at Purdue, when I learned I didn't know how to tell stories of computer science that would appeal to people in other fields.

Bob Sheldon: Say more about sparse memory. Did it use neural networks?

Peter Denning: The sparse memory was a model of human long-term memory. Its inventor, Pentti Kanerva, envisioned it as an enormous random access memory whose addresses were bit patterns thousands of bits long. Each pattern addressed a location that contained another long pattern interpreted as the response to the input. Because it is impossible to build a random access memory so large, Kanerva built it from a feasible number of locations at random points in the address space. He implemented a location as a device with an address field containing its random address and a data field consisting of a series of up-down counters. So, for example, if your pattern size was 10,000 bits, one of these devices would have 10,000 bits of address and 10,000 counters. You would build as many of them as your budget could afford.

When you present an input pattern to the memory, it selects all the devices whose addresses are within a given Hamming distance. The distance was a fixed parameter of the memory. Your input pattern would be written into all the selected devices by upping a data counter for each "1" bit of the input and downing it for each "0" bit. You could read an output pattern by combining all the counters of the selected devices, outputting "1" when the count was nonnegative and "0" otherwise.

Kanerva showed how this structure maps directly onto the neuronal structure of the brain. Although it could be implemented by an equivalent neural network, Kanerva chose this description because it was much easier to understand.

The memory could do very interesting things. Here was one experiment. The input patterns were bit map images of handwritten letters and numerals. Training consisted of showing input bitmaps and desired outputs, for example, bitmapped handwritten "A" input and ASCII code "A" output. When shown a new bitmapped "A" not in the training set, the trained memory correctly output the ASCII "A" most of the time. In those tests, the memory had about the same level of accuracy as existing optical character recognizers, not bad for a prototype.

Our group also looked at neural network models for other applications. In 1982, John Hopfield, a physicist, proposed a new kind of neural network. The NASA group interested in automatic landing of aircraft tried one of his networks. They trained it by recording the detailed motions of pilots landing an aircraft. Unfortunately, when they tried controlling the aircraft in a simulator using the neural network auto-lander, it occasionally crashed the aircraft. For that application they wanted 100 percent accuracy. The neural networks of the day could not achieve that level of accuracy.

Bob Sheldon: How long did you spend at NASA Ames?

Peter Denning: I stayed there until 1991.

Bob Sheldon: Can you say more about computational science?

Peter Denning: In 1982, physicist Ken Wilson from Cornell received the Nobel Prize in physics. He had been using supercomputers to investigate a problem in physics concerning phase changes in materials, such as how ferromagnetic materials switch polarity under the influence of an external magnetic field. He investigated this using computation to simulate the relevant mathematical model and came up with the breakthrough ideas that earned him the Nobel Prize. He was a big advocate of computational science and advocated that the government should get into very high-performance computers and make them available throughout science. Scientists started talking about "grand challenges" in science, which were very ambitious science problems that might be cracked with very high-powered computing and algorithms. One of the favorite examples is the ability to design an aircraft the size of a 777 aircraft. At that time, they projected it would take a teraflops worth of computer to do that job, which did not exist at the time. Building a working teraflops computer became a goal of DARPA and other government agencies. Boeing and others took advantage of this research.

The NASA scientists working in this area advocated that computation is a new way of doing science, different from the traditional ways of theory and experiment. The new way involved doing explorations and experiments via models and simulations on supercomputers. It also involved explaining natural phenomena as information processes and using computing to learn more about them. Computational science became a strong movement throughout science. Many fields formed a computational branch to pursue it.

The computational science movement became political. It culminated with a law in the US Congress, the High Performance Computing Act of 1991, sponsored by Senator Al Gore, which recognized computational science and supercomputing and set aside federal funding to support research in those areas.

The computational science movement also had a significant transformative effect on computer science. Computing professionals and educators had a self-story that computing was concerned with automation. Computational science focused on computing as a way of understanding, explaining, and exploiting natural information processes. That is a much broader perspective than automation.

Bob Sheldon: Then you left NASA Ames. Where to next?

Peter Denning: In 1991, my wife and I both wanted to return to the university environment. Energized from our work outside, we thought we had something to bring back. Because California universities were in a recession at the time, we looked in the Washington, DC, area. She became professor and chair of computer science at Georgetown and I became professor and chair of computer science at George Mason University. We used to say to our friends that we represented two of Washington's three academic Georges. (Georgetown, George Mason, George Washington)

Bob Sheldon: Had academia changed much in the years you'd been away?

Peter Denning: I came back as a reformer. I saw a number of ills affecting computer science and engineering education. For example, employers did not seem to trust that computer science degrees certified competence at the jobs for which graduates were hired. Employers seemed to regard degrees as evidence that a rigorous screening process had identified the greatest talent. But that only got the graduates an invitation to interview. Industry was beginning to invent its own interview processes to assess candidate competence. Interviews looked like inquiries into a candidate's skills as a programmer and problem solver. Candidates would visit with a series of groups of the employer, who would ask them to solve problems right there on the spot. Candidates took out their laptops right there and programmed something to solve the problem. If you passed all these tests, you would get an offer. This mode of interviewing was relatively new at the time and has since become very popular.

Problem solving ability was not the only thing employers complained about. Being a good member of a team and a good person interacting with customers were other major complaints. These complaints were a puzzling kind of discontent because, on the one hand, employers would say, "Your graduates coming to our organization don't know how to communicate. They don't know how to get along on a team. They don't know how to interact with customers. We are not able to put their engineering to good use, because they just don't have the sense of what good use looks like." And on the other hand, they hired every graduate we could produce and asked for more.

As I was preparing for my transition back to academia, I collaborated with my friend and teacher Fernando Flores, to produce a manifesto called "Educating a New Engineer" (*Communications of the ACM* volume 25, number 12, December 1992). We talked about what it means to be an educated engineer in the kind of world that exists and is emerging. Obviously, an educated engineer needs to be competent at engineering, but that person also needs other skills including sensibilities for how their teams are moving, the new kinds of work emerging for the digital world, and producing innovations. The list of such sensibilities and skills was generally labeled "soft skills." Academics were at best conflicted on how to teach soft skills. Some maintained that there was no room in the technology-packed curriculum for soft skills. Others acknowledged that technology is developed by teams and sold to discerning

customers. You cannot be successful without these skills. Learning the soft skills is hard work!

When I interviewed, I had in hand my written manifesto outlining concrete ideas about how to modify the curricula to develop these sensibilities. When I discussed all this in my interview with the George Mason University president, he seemed very pleased and declared, "This is a marriage made in heaven. You should come here."

I talked about my manifesto in my interview with the Computer Science department. The general reaction was, "Very interesting." In the months after I came on board, I boiled the manifesto down to a set of principles for interacting with our students. I asked the faculty to adopt these principles for our department. They voted affirmatively at a faculty meeting. I was very encouraged that the fundamental principles of the manifesto were agreed to by the department! The provost was very pleased that I got what appeared to be a commitment to reform the curriculum.

I enlisted a small group of faculty allies to make proposals for curriculum changes. When we started making specific course proposals to implement the principles, we had trouble getting the votes at the faculty meetings. Our critics said, "Don't mess around with my course; it is a good, tried-and-true course." Getting curriculum revisions through the faculty was much harder than getting the principles through the faculty.

Bob Sheldon: They had their rice bowls.

Peter Denning: I suppose you could say that. They were protective of the status quo, especially when a proposed change would alter the part of the world they were a specialist in. It was very disappointing to have the sense that they agreed with principles, but they didn't want to implement the principles.

Bob Sheldon: Was George Mason University's Computer Science department in a big growth mode then, because the university overall had a big growth spurt?

Peter Denning: George Mason was in constant growth and that continues to this day. When I came, there were fewer than 20,000 students; today the enrollment exceeds 35,000. Our computer science and engineering student body was steadily growing. After I left, the administration decided to merge the computer science and information systems departments, to better accommodate the student loads and develop an integrated curriculum. Now they have a large department.

Bob Sheldon: Did you formulate your "great principles" while you were at George Mason University?

Peter Denning: I started to work seriously on that around 1997 at George Mason. The seeds were planted, as you recall, in my time as department chair at Purdue when I was engaged in the question of how to have computer science be more interesting for people. My time at NASA nourished those seeds in the NASA soil. After my involvement in the formation of computational science, I became very interested in refuting the claim that computer science was not really a science. That old claim was still alive because some people still resisted the idea that computer science could be its own field.

I set out on a project to articulate what I called the "core principles of the field." I believed that a good articulation of our core scientific principles would go a long way to refute the claim that computer science is not science. All the other fields—physics, astronomy, chemistry, all of them—do that. Why not us? I found many sympathetic colleagues who wished as I did that we could give a good account of the core principles of our field.

Doing this turned out to be harder than I imagined. I asked many people for their opinions. "What do you say the core principles of our field are?" Typical answers would be: "Programming languages, databases, and operating systems, and networks, graphics, robotics, and a few more." Echoing our critics, I said, "Those aren't principles. Those are technologies. What were the principles the technologies were based on?" It took a while to tease those things out. I often had conversations that went like this: "What is a core principle in your area?" Frowns and scowls, and finally a question, "What do you mean by a principle?" I said, "You're the expert in your area. What are the principles you think are really important? Principles that will still be here years from now when the current technology is gone?" They hadn't actually asked that question before. They were teaching mostly big ideas and clever

technologies. I probed further: "What are the deep, cosmic scientific principles you're working with?" They hadn't thought about that. My work turned into getting people to think about that question and give voice to the deeper, timeless scientific principles they deal with, principles that keep showing up in each generation of technology.

An example in my own specialty, operating systems, is the principle of locality. This is the principle that computations tend to refer to their code and data in relatively small subsets over relatively long periods of time. Since the code and data are stored in memory, this principle leads to the design of efficient and fast memory hierarchies and caches, including Internet caches. Every generation of operating systems experts has investigated this principle and found it in all their computational workloads.

Bob Sheldon: Did you bounce these ideas off other friends in the ACM community?

Peter Denning: Yes. I organized a "great principles task force" of the ACM education board, a group of about 30 or 40 people to collect proposals for statements of the great principles of computing. We put together quite a list. Some proposed principle statements didn't sound like deep principles and were discarded. Others had that "timeless, cosmic" feel to them and survived. Even after all our culling, we still had quite a list, around 60 proposed statements. I noticed they could be grouped into six categories: communication, computation, coordination, recollection, evaluation, and design. These were not mutually exclusive, but instead were important perspectives on computing. I brought all that to NPS along with me in 2002, and with my colleague Craig Martell I designed a course for students called "Great Principles of Computer Technology." Our students found this course to be a very helpful introduction to computer science.

Bob Sheldon: So that's when you came up with the textbook *Great Principles of Computing*?

Peter Denning: We wrote that book on evenings and weekends in our personal time. The course didn't call for us to write a textbook.

Bob Sheldon: Was there general consensus in the ACM and professional community about these principles?

Peter Denning: We had consensus on that ACM education task force. We had to blend it in with some strong forces in the education world. Let me say a few words about the gathering forces so that you can see where this goes next. For years, computing educators have tried to get computer courses into K-12 schools, arguing that computers were becoming so ubiquitous that every child needed to know something about them. In the 1980s, university educators worked with middle and high school teachers to get a "computer literacy" course into their curricula. The result was courses about how to use common computer tools such as document preparers and spreadsheets. The teachers did not have the computing knowledge to teach anything about computing principles or programming. The literacy courses were not successful. Children got bored with them and learned nothing about computing principles or programming. Literacy did not work out well.

In the late 1990s, some of our education colleagues persuaded the National Academy of Engineering to sponsor a study, which came to be known as Fluency in Information Technology (FIT). Their recommendation was that literacy was too low level, and we should be going for fluency—the ability to "speak the language well" and do useful things with computing. The leader of the panel, Larry Snyder, subsequently wrote a book *Fluency with Information Technology*. It became very popular; it's used, I think, by a lot of high schools and colleges. It's now in its sixth edition. Fluency got us a bit further with more penetration than literacy. But it didn't turn into the movement that educators were looking for.

In 2006, Jeanette Wing wrote a short essay as an opinion piece in the *Communications of ACM* called "Computational Thinking." She proposed that instead of the terms fluency or literacy, we use "computational thinking." She argued that everyone wants to put computers to good use and they need to think like a computer scientist to do that. Computer scientists could teach K-12 teachers how to think about computing, thus empowering them to provide computer education for their students. Educators who were frustrated with previous attempts to get computer courses into K-12 schools found this proposition attractive. Wing went to the National Science Foundation (NSF)

in 2007 and started to mobilize NSF staff and resources around computational thinking. They defined a series of initiatives that brought a large number of educators into a computational thinking movement. These initiatives included building a computational thinking component into most research projects, sponsoring professional groups to define computational thinking curricula for K-12 schools, training 10,000 K-12 teachers in computer science, and developing a new advanced placement (AP) curriculum and means to interface it with universities. In all, NSF put about $48 million into these initiatives, which were quite successful.

The AP curriculum in high schools culminates in an AP exam. Students who pass the exam get credit for the introductory computer science course at the university they select. A version of the AP curriculum approved in 2001 emphasized object-oriented programming and the Java language. Teachers did not understand the complexities of object-oriented programming and had difficulty teaching it. It turned into a disaster with dramatically fewer students enrolling in the AP curriculum for computer science. The NSF initiative to reform the AP took the form of support for a task force of the Educational Testing Service to advise them on a better curriculum, and the creation of a new kind of university first course, called "CS principles," that interfaced with the new AP curriculum. Several of the people who had participated in the great principles task force designed CS principles first courses at their universities. Each transformed the general idea of timeless principles into a course for their students. This is how the earlier work on great principles affected the design of the new CS principles courses.

As the teachers' professional societies began to weigh in with proposals for K-12 curricula based around computational thinking, I became very concerned. They were formulating computational thinking as basically programming—how do you construct a program that meets a specification. They didn't even use the word "design." The objective of programming became expressing solution procedures as computational steps; that was said to be the heart of computational thinking. This just didn't square with everything I've learned about computational thinking. It was way too narrow. There was no engineering in it, for example. We have to build real computers out of real circuits and real components to do real things. And they're not even talking about that. They're just talking about formalizations like modularization, abstraction, recursion, data structures, and other aspects of programming. They're not talking about how you design circuits of machines that do these things. Or what's the right relationship between a program and a machine. Or how to deal with issues of large systems.

There was nothing in these formulations about design. Writing programs is generally a bigger issue than meeting precise mathematical specifications. It is to create a world for users of software and programs, in which they can get jobs done of value to them. I began raising objections.

Bob Sheldon: Was this when you were at George Mason University?

Peter Denning: No, this started in 2007. I left George Mason in 2002. My work at Mason focused on defining the fundamental principles of computing. At that time, we were not paying much attention to computational thinking. If somebody were to ask at that point what computational thinking was, I would have answered with some generality such as "the mental skills of designing computations and computing systems to do jobs for us."

The computational thinking movement began in 2007. Its main purpose was to get computing into the curricula of K-12 schools. I endorsed that purpose. However, in that movement I saw misconceptions emerging that would undermine the purposes of the movement.

First and foremost, I don't know why the activists in the movement wanted to ignore the history of computational thinking. Yet they did. The history of computational thinking goes back 4,500 years! The algorithms of yore were precise instructions for human beings to carry out numerical calculations. The mathematicians and engineers who used them were thinking computationally. In short, computational thinking has a much bigger tradition than the proponents were making out. They were unknowingly misrepresenting computing by

portraying it as a much narrower field than it really is. Even worse, in my mind, the educational recommendations for K-12 curricula incorporated a number of serious misconceptions about computing. If we do not correct these misconceptions, we're going to graduate generations of students who think they can get computers to do things that are impossible. That's the essence of my concern.

The misconceptions sparking my concern have not ameliorated. A couple of years ago I wrote up a piece called the "Remaining Trouble Spots with Computational Thinking," formulated to not sound too critical, but point out major omissions in the computational thinking juggernaut. It seemed to resonate with a lot of people.

I've just completed a book, *Computational Thinking*, with co-author Matti Tedre, which will be published by MIT Press in Spring 2019.

Bob Sheldon: You said you left George Mason University in 2002 and came to NPS. What motivated the move?

Peter Denning: My wife Dorothy and I really like California. We always had an objective in the backs of our minds that if the right opportunity came up, we would return to California.

In spring 2002, Cynthia Irvine, a professor here, asked Dorothy to write a letter of reference for a faculty candidate. Dorothy said to Cynthia, "You have open positions. I didn't know that!" Cynthia caught that one and said, "Are you interested?" And Dorothy said, "You bet!" That opened a conversation that led to interviews and eventually to a pair of offers. My offer was to join the Computer Science department as chair. Hers was to join the Defense Analysis department, where she could work on cyberspace security issues in the defense domain. Not only did these offers invite us to work on issues of great interest to us, they brought us back to the West Coast, in the Monterey area that we always liked.

Bob Sheldon: You were able to keep up with your research on the great principles when you moved out here. Did you delve into other lines of effort when you moved out here?

Peter Denning: I resurrected my interest in memory management in computing systems and partnered with the Locality Research Group at University of Rochester, helping out with several student theses including being an external member of a PhD committee. This has been also useful in our course on operating systems.

A big question that I grappled with at MIT and down through the years after was how to arrange programs and data in the memory hierarchy to get the best throughput from the system. Consider an example to see why this is important. Suppose I want to decide when to keep a page of data in main memory or on the hard disk, which is 1 million times slower. After I just used the page, if I could see exactly when in the future it would be used again, I could do a simple calculation comparing the cost of keeping it in memory until next use (rent) with the cost of removing it now from memory and paying for its retrieval later (swapping). Even though I can't see into the future, I can approximate this ideal by appealing to the principle of locality. If my page was used recently, it's likely to be used again shortly, so I pay the rent and keep it. If not used recently, it's likely to not be used in the near future and I'm better off removing it now. If I apply this strategy to every page loaded in memory, I can get the system throughput to be very close to what it would be if I had infinite memory.

Long ago I invented a way to realize this strategy, called "working set." The working set of a program is all the pages it used in an immediate past sampling window. I run memory management to detect working sets and protect them from removal while their programs are running. The theory, confirmed by experiment, is that working set detection and memory management yields close to optimum system throughput.

The Rochester Group has been exploring how to exploit the principle of locality to manage the cache memories that pervade every computer system. Doing it with methods related to the working set optimizes cache performance.

I have been finding that my students have trouble understanding the principle of locality. It seems counterintuitive to many of them that programs have such pronounced locality behavior, and that tracking it optimizes performance. Maybe it's because I've been so familiar with this over the years, it is obvious to me, but not to newcomers.

Bob Sheldon: OR people would listen to your explanation because optimizing is a big part of our background.

Peter Denning: A different question, what is innovation leadership, has become a central focus of mine at NPS. The Defense Department has made very strong statements about being better at innovating. Some seeds planted when I was at George Mason University in the early 1990s seemed to offer a new answer to this question.

In 1993, I developed a course called "Sense 21," shorthand for "a new engineering common sense for the 21st century." It was related to the engineering education manifesto we discussed earlier. The word "sense" in that title means that we need to learn a new "common sense" about how the world works if we are going to be successful engineers and innovators in a world of accelerating change full of contingencies and surprises. The prevailing common sense is that we find recurrences and then try to control events and people to exploit the recurrences and attain the desired outcomes. The new common sense is that we become navigators who can skillfully blend with surprises and contingencies to move toward the desired goal, and we mobilize people to follow us. In other words, recurrences and control give way to navigation and mobilization. I started Sense 21 because my students had important questions about how to function as engineers that were not being addressed in the standard curriculum. A common question was time management. How do they deal with demands of their day jobs, their evening classes, and their families within limited time constraints? Another was finding innovative ideas. How do they find them? Having found them, what should they do when someone else's stupid idea is selected instead of their good idea? How do they deal with unruly customers who do not understand the technology they are asking for? How do they work effectively on a team, especially when there are conflicts? Buried inside all of these is the concern that "They want me to be a better innovator, but they don't tell me how to do it, and I don't know how to do it. I'm being held accountable and I don't know how to do it."

The Sense 21 course was about how to generate innovations. It offered a "new common sense" about what shapes action in the world and how innovation works as a skill. Our new common sense was that "innovation is new practice adopted in a community." Along the way, the students found answers to their pragmatic questions that I just mentioned. They all produced small innovations by the end of the course and saw that later, with more experience and skill, they could produce larger innovations. They resolved many of their work issues that motivated them to come to the course. They found the course transformative.

At the end of the first offering of that course, the students asked, "Are you going to have a follow-on course, like Sense 21 part 2? What comes next?" I said, "We have nothing designed." And they implored, "We have to have something! You can't leave us stranded!" They wanted more because what they had already learned changed their lives. Wow, this is deep stuff. To accommodate their request to continue their learning, we formed an alumni group. We met monthly in the evening over pizza and talked about topics the students selected, always true to the interpretation that innovation is a new practice adopted by a community. This alumni group continued for the next 10 years, finally disbanding when I left Mason in 2002. Parenthetically, I would add that this is the only course I ever taught where the students wanted to form an alumni group. Nobody from my operating systems courses wanted to form an alumni group.

Bob Sheldon: Can innovation be taught at a university?

Peter Denning: It can. My starting point—innovation is adoption of a new practice in a community—is completely different from the more familiar notion of creating new ideas or inventions. But it leads to new insights into how innovation works and how you as an innovation leader can facilitate or shape it. What is the skill set that enables me to be a successful innovation leader?

Over the years of looking into this question, I learned enough that I was able to write a book. Robert Dunham and I wrote *The Innovator's Way*, published by MIT Press in 2010. In that book we talk about why popular theories of innovation fail to consistently produce the adoption outcome. We discuss at length the eight essential

practices written on the cover of the book: sensing, envisioning, offering, adopting, sustaining, executing, leading, and embodying. We called them essential practices because if any one of them fails to produce its intended outcome, the whole innovation is likely to fail.

Bob Sheldon: Is that used for an engineering course here at NPS?

Peter Denning: No. Marine Corps Colonel Todd Lyons and I have designed a course now called "Innovation Leadership." It is an adaptation of the Sense 21 course I taught at George Mason from 1993 to 2002. The course has really helped our students shape their innovation projects, which are likely to span many years of their careers beginning with their thesis here. At the end of the last version our students gave presentations of their innovation projects. All were TED quality. I mean that if any one of them stood on the TED stage, they would move the audience with their big ideas just as much as the professionally produced TED talks. All these students found something deep inside their own histories that gave them a sense of destiny and a determination to pursue their project for years beyond their NPS thesis. In the course debrief at the end, the two most common comments we get from students are: "This course changed my life," and "I learned I can intentionally produce innovations, starting first in small groups and expanding later to large ones."

Bob Sheldon: Was this a master's level course?

Peter Denning: Yes. This is an elective course. Colonel Lyons and I screen the students who want to join to be sure they are truly interested in the outcome we promise and will be committed to doing the work on the course to get it. This course is about leadership practices for intentionally producing innovations. It is not about generating ideas for innovation. Most organizational leaders say they are honestly overloaded with lots of proposed ideas, but they have to figure out which ones are worth spending their limited energy and resources. Organization leaders say they have no shortage of ideas, but rather a shortage of people who know how to transform ideas into practice. That's where we come in.

Bob Sheldon: You were at George Mason University and then here at NPS. At George Mason University almost all your grad students worked during the daytime and went to school at night. Here at NPS, going to school is a full-time day job. How does the relationship between the students and the instructor compare in those two environments?

Peter Denning: There's a world of difference. It took me a couple of years to figure it out. In 2004, I realized that the core of my relationship with Mason students was markedly different from NPS students. At Mason, I was the computer science guru, a gatekeeper whose approval was needed if they were to get the degree and enter the field as a professional. Their job was to convince me, the gatekeeper, that they're worthy of passage through the gate. At NPS it took me a while to uncover my hidden assumption and learn that NPS students do not see me as a gatekeeper; their profession is already chosen—military officer—and my job is to give them computer science skills that will enable them to be better officers. That's a different job from being a gatekeeper. Once I learned this, my interactions with students were much better.

Bob Sheldon: Another difference is that at George Mason University, the classes are typically one night a week for three hours, and here at NPS you have them three days a week for an hour.

Peter Denning: In my Computer Science department, we have a lot of courses with four hours of lecture and two hours of labs—we're seeing those students for six hours a week. Most of these students have four courses. From their perspective, each day is packed. Most days, they'll see three of their four courses, and other days they'll see all four. They've got homework to do for all these courses. And they're trying to balance that with family life because they have kids. These students have many of the same problems that these George Mason students had about time management. They also have concerns about chain of command, often believing it is better to shut up and say "Yes, sir!" than to ask a more senior officer a question. They also have a big concern about good grades since their performance reports depend on that. I find myself helping students with many similar, non-course issues as the Mason students. They have to break through these taboos if they are to be innovation leaders.

Bob Sheldon: Since you came from a non-military background yourself, and now you're in a military academic department, how has your view of the military changed?

Peter Denning: I had basically no contact with the military before I got to NPS; I came with an open mind. I developed a lot of respect. A lot of the stereotypes of the military didn't apply to what I was seeing. The military stereotype might be that the military are bent on trying to quietly subjugate society to the military way of thinking. Or that they are itchy to use military power and are annoyed by politicians reluctant to do so. Instead, I learned that our officers and their military leadership are very wary about getting involved in wars. They see the magnitude of destruction and disruption and don't want to do that unless absolutely necessary. It's the politicians who itch for war. For example, when first asked to enter the Iraq war, the military advised the president, "You have a bigger objective here. You want Iraq to be a democracy. That's not a military mission. We can certainly go in and knock out Saddam Hussein, which we did, but we can't deal with all the rebuilding. That's not our job. You want us to do this job, but it's not a job we know how to do." The military is extremely cautious about getting involved in these things. In the end, it's the politicians that say, "Do it." So the military has to do the best they can.

The military is also interested in the big issue of innovation. They face innovative and agile adversaries who are not part of governments and who are very clever about leveraging technology to undo the US military asymmetric advantage. The military wants to be able to counter them. They don't want to lose their military advantage. They want our officers to become much more agile and able to get innovations in place despite the bureaucracy. Everybody says the military bureaucracy is stifling, but military people have a way of getting around the bureaucracy when they have a genuine need to do so. So innovation inside the military is a big concern. They see there is a limit to how much they can accomplish simply by purchasing technology through the very slow acquisition process. They have numerous examples of great technologies languishing in warehouses because no one knows how to use them. In other words, they want adoption—not simply the delivery of devices. We're trying to help them with that.

I should add that the military is quite innovative already. Their concern is to be more innovative than adversaries so that the US military advantage is not eroded. They're not warmongers. We would actually probably be involved in more wars, except for military pushback not always publicly visible to avoid conflicts that cannot achieve a clear win.

Bob Sheldon: Could you comment about how the Computer Science department at NPS provides value-added to NPS, to the Navy and the Marine Corps?

Peter Denning: Inside the department we have several tracks. They're actually areas of specialization. They are cyber security, networking, artificial intelligence and data science, modeling, virtual environments, and simulation (MOVES), and software engineering.

The cyber specialization, one of our most popular, deals with cyber defense and some cyber attack. Cyber defense and attack go together pretty strongly, because to defend you need to know how the adversary is likely to attack. A lot of students are involved in that, trying to become cyber warriors. The Navy Cyber Command asks us for a very strong technical program because they have come to understand that most of the attacks we're trying to defend against are happening down at the lowest levels of the operating systems and networks. You really need to understand the operation of the system at those low levels to figure out how an attacker is working. This requires a good deal of technical expertise with operating systems and networks. We're trying to provide that. Admiral Tighe (who served as Deputy Chief of Naval Operations for Information Warfare) believes that a leader of a defense team cannot be a good leader unless they understand the technology they're trying to defend. It's not enough to have an MBA-type degree. Our operating systems, networking, and low-level programming courses give the students the depth they need to be good leaders of successful defense teams.

The networking area is another popular specialization. Marines particularly like that one. They are very interested in mobile devices

for field use and the security of the networks of these devices. And they want powerful apps that can be put on these devices to support field operations.

Our artificial intelligence track responds to the Navy interest in unmanned, artificially intelligent systems. We get a lot of inquiries for education on artificial intelligence and machine learning and the related area of data science. Our students work with other departments on this area. For example, the mechanical engineering department has an active drone research program to which some computer science students have contributed.

Our MOVES track provides students with expertise in vision, decision support through visualization, training environments including games, and augmented reality.

Our software engineering track focuses on the design and implementation of large software systems that are reliable, dependable, usable, safe, and secure. This is an advanced topic that deals with systems, not individual programs. We work closely with the systems engineering department because most of the applications of software engineering fit in that context. When I first came to NPS, we had a software engineering master's degree, but the Navy stopped sending students to it. That decision always puzzled me because almost all systems today incorporate software. I'm a little disappointed that the Navy hasn't sent more students to us to learn software engineering.

Bob Sheldon: Is software reliability taught as part of computer science?

Peter Denning: Yes, in several places. Our software engineering covers reliability of software systems. Our cyber courses cover secure coding practices. Reliability is significantly complicated in software that is so big it requires teams to develop. Our lower-level programming courses discuss reliability as the meeting of specifications of a program. Often mathematical methods such as correctness proofs, or testing methods are sufficient for these small programs. It's the large programs and large systems of programs that are so difficult. Reliability is a big concern of systems engineering, with whom we collaborate.

Bob Sheldon: I know that "R"—the free software environment for statistical computing and graphics—is used for some of the courses here at NPS, and open-source software is becoming popular. What are your thoughts on open-source software?

Peter Denning: Linux is a widely used open-source operating system. It's been worked over by thousands of programmers and is very reliable. But it's also very large. The main Linux distribution is something like 420 million lines of code. Literally, it's all those thousands of volunteer contributions tied together, and it's not structurally optimized in any way. It's extremely complex, even though it's reliable. By comparison, the Apple Mac OS system and Microsoft Windows are 50 to 80 million lines of code. Those two operating systems are less complex than Linux. They are much more reliable than their predecessors but still suffer instabilities and need for frequent security patching. There are numerous security stresses including malware attacks, botnets, denial of service, social engineering, and more. It's hard to tell whether Linux is more resistant to security attacks than Windows or Mac OS.

Bob Sheldon: For the past several years, I've read about the shortage of pure computer scientists, because there's a big demand for them and not too many students going into it. Is computer science having trouble selling itself to the younger generation?

Peter Denning: There has been little problem in the last few years with marketing computer science. Everybody seems to be on board with it. Across the country and the world, students are flocking to computer science majors and many departments are engaged in very aggressive hiring of new faculty to meet the teaching need. This has happened because everyone now sees there is an ongoing computer revolution, which is producing amazing technologies such as the cloud, and many students see a future for them by learning to be builders and designers. The growth of cloud platforms now enables almost anyone to get massive amounts of computing power and storage quite cheaply. There has been an explosion of startups with many new ideas of how to exploit the new technology with new products and platforms. We also are awash in buzzwords including not only the cloud, but also artificial intelligence, deep learning, data science, social media, and blockchains. Young

people want to be part of the action. I don't see a loss of demand for their skills in the near future. I see a widespread acceptance of computer science. Employers still complain, as they have done for years, about lack of soft skills in our graduates, and yet they hire nearly every graduate. The companies have their own unique ways; once the new people are brought in, they seem to do well developing soft skills such as teamwork and collaborating. I don't see a backlash of any kind against computer science.

Bob Sheldon: A sort of wrap-up question. Where do you see the field of computer science evolving over the next few years?

Peter Denning: That's always hard to say. A computer revolution is definitely under way, but it's hard to say which of the many possibilities now emerging will become a big thing. I like the term "avalanche" for a large-scale economic-social force that is beyond anyone's control and can sweep away familiar ways of doing things. The Internet was a past avalanche—look at all the things it has changed since we were young professionals. The cloud is an avalanche-in-progress. Amazon has a multibillion-dollar industry—Amazon Web Services—which is their cloud. It's huge, all over the world. Google, Microsoft, and Apple are all doing similar things. Many new small players can now access massive computing very cheaply. The cloud libraries are getting better and better—powerful tools for speech recognition, image recognition, big data analytics, modeling, and simulation are now available. All these things are transforming the way people approach getting into the computer career field and using computers. Of course, there's a dark side to transformations, such as mass surveillance and massive data collection threatening privacy. All that is part of the big forces gathering around the cloud avalanche. It will all have a very significant influence on how the field turns out. We don't know how many of our current questions are going to be answered, but we do know the status quo is not going to survive.

I doubt that the cloud is the only economic avalanche that will reshape our world. We need to be better observers of the precursors of avalanches and take steps to prepare.

We have already noted that previously technical terms such as artificial intelligence, big data analytics, deep learning, and blockchains are all highly hyped buzzwords. Everybody seems to be jumping on someone's bandwagon. A lot of claims are being made that can't be substantiated. We'll see how that plays out.

Consider big data. Ever since I was a graduate student there's always been a big data problem in every generation of computers I've ever seen. There's always data around, bigger than whatever computer you've got. Now we've just come to the point with fast worldwide Internet connectivity that we can now dream of the whole planet as a computer and the whole world as planetary-size big data. Some of the things that are being done with big data are pretty amazing, and some of them are downright scary and creepy. How is that going to play out? I don't know. Trying to read how all this is moving is like earthquake preparedness. Everybody hears about it all the time, but few people are actually prepared. Avalanches are like that. They might come, but day after day nothing happens, and we become complacent. Then suddenly one day it lets loose, and we find how few of us are actually prepared. Not only are a lot of people unprepared, governments aren't prepared either. As artificial intelligence is accelerating automation of some jobs, many people are getting nervous about being displaced. Governments don't have good educational safety nets in place to help people transition to the new jobs. This creates a lot of social discontent. As computing continues to push forward and accelerate the pace of change, it will foster more avalanches and disruptions with large-scale economic fallout. Many of the new problems that appear won't be solvable by computing technology. They will be social problems. We've all got to learn to live together to deal with these things.

Consider also blockchains. Ever since the bitcoin crypto-currency appeared in 2009, its underlying technology of a distributed ledger has attracted huge investment and massive public attention. The price of a bitcoin skyrocketed to $20,000 in late 2016 but has plummeted to $3,500 lately and appear to be headed lower. The advocates of blockchain say bitcoin is not the source of the value; the blockchain is.

However, there are serious questions about blockchain. The cost of adding a new block to the chain is purposely set very high to prevent fraudsters from tampering with previous blocks. The cost of synchronization messages among servers holding copies of the blockchain is high. The databases stored in a blockchain are typically very, very large and the full blockchain cannot be accommodated on a single server. Established database companies such as Oracle have other, faster, and much cheaper ways to implement distributed databases. If an avalanche is forming, it might backfire and wipe out the blockchain companies, or it might lead to more distributed databases that are not implemented as blockchains. It is hard to say how all this will play out.

There will certainly be avalanches involving computing but they are unlikely be the ones featured in the buzzwords.

Bob Sheldon: Any other parting shots?

Peter Denning: Sure. I have been mulling an issue that concerns the health of computer science and engineering in the years ahead. Basically, I think we have come to be dominated by a science interpretation of what we do and have lost touch with an engineering interpretation. For a healthy computing field, we need both, and they must be in balance.

As recently as 200 years ago, engineering and science were not distinguished from each other. They were seen as technology. Many of our hero stories, for example the Wright brothers and Edison and others, come from a time when science and engineering were not distinguished as separate fields.

Beginning after World War II, we started making the distinction, in the United States anyway, that science was separate from engineering. Vannevar Bush, the President's science advisor right after the war, was the main articulator of the distinction. He advocated the formation of the NSF to explicitly support university research in science. Engineering was part of NSF but was downplayed. What we call the science interpretation today grew from this start and became dominant.

Fundamentally, science is looking for recurrences—repeatable patterns—and then showing that they're sufficiently reliable that you can use them for explanation and prediction. We call them laws when we find them. The science interpretation also places a high value on the "dispassionate observer." This observer is always situated outside of whatever they are observing and aims to be objective and unemotional.

The engineering tradition, which predates science by thousands of years, is almost the opposite of science. Egyptian engineers built the pyramids; Roman engineers the aqueducts; English engineers the great steamships. Engineers have always been involved in their communities trying to solve pragmatic issues using technology as the means. The engineer is not outside the action; the engineer is part of the action. The engineer is not a dispassionate outside observer of the community; the engineer is passionately involved in the community. The historical engineer often does not "apply recurrences"; the engineer harnesses effects that might become recurrences in the future. The historical engineers learned to care about the concerns of their clients. The modern scientist learns to be dispassionate and uninvolved with clients. I'm making a distinction between the historical engineer and the modern view of engineer as an applied scientist.

The modern definitions of engineering say that "engineering is the application of science to solve useful problems." According to this definition, engineering is a subset of science! Engineering curricula have adapted to this idea and teach engineering as if systems and artefacts can be designed without an understanding of the concerns and interests of their customers. Systems and artefacts are objects that meet specifications produced by disinterested observers.

The historical engineer is not a disinterested observer, but is rather a very concerned and involved and interested observer. I've portrayed science and engineering in a stark way here to make the distinction. In practice, science and engineering cannot get along without each other.

Consider how the different interpretations approach contingencies—those unanticipated events that pop up unexpectedly and derail a project. Scientists approach contingencies by giving them names (e.g., "black swans"), building mathematical models (e.g., "power laws"), and developing theories (e.g., "chaos

theory"). At best that explains a contingency but does nothing to resolve it. Engineers respond to contingencies by inventing patches and workarounds, and then having after-action meetings to determine how to approach the issue in the future. Clearly this common practice of engineering is not science or the application thereof.

For me, the notion "engineering is the application of science" is just wrong. The dichotomy between "searching for recurrences" and "harnessing effects and handling contingencies" is false. Engineering and science have always worked together. They can't live without each other. Engineers have often discovered phenomena that could be harnessed in the engineering sense before anybody fully understood the phenomena. The scientists helped to understand it, came up with the explanations and the laws, which the engineers could then use to optimize. Many of the great discoveries of science started with engineers trying to deal with pragmatic issues; the scientists came in later. The examples of science preceding engineering are generally building machines to confirm scientific predictions, such as building apparatus to measure the bending of starlight around the sun as Einstein predicted. Science and engineering are not the same. It's unhealthy for us to entertain the notion that engineering is a subset of science. Science, with its stress on objectivity and abstractions, is not enough to deal with a lot of the pragmatic technology issues that need to be solved.

When I talk about educational reform, I look for a better marriage between science and engineering. It's not just an abstract mathematical exercise of coming up with a computational procedure. It's trying to get a machine to do something useful for people and not screw it up. I worry about the current trend toward more science and less engineering in computer science. Computer engineering is a separate field, and computer science is trying to organize itself like it's an abstract mathematical field. I think it will peter out if that's the way it really sees itself, because in the end, if you can't do something useful with a computer, no one is going to be interested. So I would like to see our future in computer science appreciate both the science and the engineering side, willing to be involved in their community, listen, design, and all these things, find pragmatic solutions to problems. Not try to deal with everything as another programming problem.

That's my parting shot.