Interview with Barbara Liskov
ACM Turing Award Recipient 2008
Done at MIT April 20, 2016
Interviewer was Tom Van Vleck

Interviewer is noted as "TVV"
Dr. Liskov is noted as "BL"
?? = inaudible (with timestamp) or [ ] for phonetic
[some slight changes have been made in this transcript to improve the readability, but no material changes were made to the content]

TVV: Hello, I'm Tom Van Vleck, and I have the honor today to interview Professor Barbara Liskov, the 2008 ACM Turing Award winner. Today is April 20th, 2016.

TVV: To start with, why don't you tell us a little bit about your upbringing, where you lived, and how you got into the business?

BL: I grew up in San Francisco. I was actually born in Los Angeles, so I'm a native Californian. When I was growing up, it was definitely not considered the thing to do for a young girl to be interested in math and science. In fact, in those days, the message that you got was that you were supposed to get married and have children and not have a job. Nevertheless, I was always encouraged by my parents to take academics seriously. My mother was a college grad as well as my father, so it was expected that I would go to college, which in those times was not that common. Maybe 20% of the high school class would go on to college and the rest would not. I was always very interested in math and science, so I took all the courses that were offered at my high school, which wasn't a tremendous number compared to what is available today. The most advanced course that was offered was a pre-calculus course, which I took in my senior year.

There weren't many girls in these classes. I don't really remember how many there were. I might have been the only one. But that didn't seem to matter much. But I did feel from the students around me that this was not something I should be doing, so I kept a low profile. I think that I was probably encouraged by the math teacher that I had in my senior year, though I don't remember anything specific. But I certainly don't remember picking up anything negative.

I definitely was encouraged by my parents to do well in academics, although not anything specifically having to do with math and science. On the other hand, there were also no negative signs. I think that probably what was particularly important was my father, who had very high academic aspirations, and my mother who did never said anything negative about what I was doing. I think that sometimes mothers can have a lot of impact on young girls, and if they show signs of disapproving of the path you're taking, that could be a big push in the opposite direction.

My father insisted I take Latin because he said it would serve me in good stead, which it has over the years, but he also insisted that I take typing. The rationale there was that if, heaven help me, I ever had to support myself because I didn't get married or something happened to my husband, then I could get a job as a secretary. A secretary or a teacher were sort of the acceptable professions for a woman who had to work.

In those days, if you went to high school in California and you had I think a B+ average, then you were guaranteed a slot at the University of California. The question was which of the campuses would you end up going to. The top campus was UC Berkeley, and I had a very high average, so I got into UC Berkeley. I went there for college. And I started off thinking I was going to major in physics. I think this was the aspiration of many students because physics was considered to be the hardest major. But I pretty quickly discovered that I didn't have a lot of aptitude for physics and I liked math a lot better, so I switched to a math major and a physics minor. I didn't know anything about computers. I don't remember any computers as an undergraduate. They were there. I mean I've talked to people after the fact and I know that some of the men who were there at that time had discovered them. They were probably in the School of Engineering. And Berkeley has a separate admissions process for that. I did not apply to engineering. It never crossed my mind to do something like that.

So I majored in math in the College of Letters & Science. Again, looking back at these classes, if there was another woman in the class, that would be it, and often I was the only one. And I would say that at Berkeley, I definitely got push-back. You know, I was not the one called on in class, I was not the one that was invited to do a research project with somebody. I was usually the top or one of the top two in the class, but it didn't really make any difference. On the other hand, I never encountered anything overt. Or if I did, I didn't notice it, because I have a tendency to not pay too much attention to negative signals, which I think has stood me in very good stead.

So I majored in math, I minored in physics, and when I finished at Berkeley, I thought about going to graduate school and I actually applied to both UC Berkeley and to Princeton. I got into Berkeley. From Princeton I received a postcard saying that they didn't admit women to their graduate program. This is 1961, so it was before the Ivy League men's schools had meshed with the women's schools, and women were not allowed into these schools. But I was very surprised. I had no idea this applied to the graduate school as well and the undergraduates

But I decided that I wasn't ready to go on to graduate school because I didn't feel like I was ready to work that intensely on my studies. So I decided instead to take a break. And I wasn't really thinking of it as a break. I was just thinking I wasn't ready to go to school. I wasn't thinking, "Well, in a couple of years, I'll go to school." I just thought this wasn't what I wanted to do right now.

My father came from the Boston area. Actually he grew up in Portland, Maine and then went to Harvard and his family moved to Boston, so I had relatives in the Boston area and I thought it would be nice to go to Boston for a while and see what it's like. I had a friend from high school who'd gone to Stanford who was interested in doing this too. She was a biology major. So we decided to go to Boston together.

We went to Boston in the summer of 1961. I didn't have a job lined up. I decided I would just go and then I would look around to see what I could find. I got to Boston probably in August and sent out résumés to various places. I wasn't able to get an interesting job as a mathematician but I was offered a job as a programmer at the Mitre Corporation, and so I took that. That was my first intimation that there was such a thing as computers. And at that time, since there were no computer science programs and nobody coming out of college who knew anything, or they were very rare, they would take anybody they thought might have an aptitude for programming.

I got a job at Mitre and on my first day at work, they handed me a FORTRAN manual and they gave me a problem. They said, "Write a program to do" something. I forget what the "something" was. I discovered that I really enjoyed this. So I'm totally self-taught as far as programming is concerned. I had to do this by myself. Nobody was training me. If I had a question, I could go ask somebody, but basically I was doing it all on my own. I discovered I really liked it. I was really good at it. I had an aptitude for it. So that was a great door that opened for me, to find something that I could do really well and that I really enjoyed.

TVV: Were there other women at Mitre working with you at the time?

BL: There definitely were other women there. There were a large percentage of women, because, as I said, they were taking them if they thought they could do it and had the ability to think logically. You didn't even have to be a math major, though math would be a good background for this. But it had to do with being able to think logically and be organized. Yeah, so I definitely remember women who were working there.

They had different levels of employees. I was just a "programmer" but there was a higher level known as "tech staff." While I was there, they hired a man and a woman, and both had master's degrees. The woman was hired as a programmer and the man was hired as a tech staff. I noticed this and thought "Hmm, that doesn't really look quite right to me." Although in retrospect, the man had a degree from MIT and the woman had it from someplace else, so there could have been a rationale. But that was how things were in those days. In fact, to get that job, I had to tell them that I was looking for permanent career employment. That was the way you had to approach these jobs, because they were worried about women coming and them leaving, and they would have lost their money or something.

I worked at Mitre for a year and I was living in Cambridge. I saw an ad for a position at Harvard working on the language translation project and I thought it would just be nice not to have to commute, so I applied for that job. I got that job at a pretty good raise in salary. I knew nothing about how you might negotiate for a higher salary by getting a competing offer. Mitre offered to counter that. They offered to give me a tech staff position. But of course it wasn't why I was doing it. I just thought it might be fun to do something different for a year. So I went to work at Harvard on the language translation project.

That was a good move as it turned out. The project used a huge program that was written in assembler - it was probably for the IBM 7094. I think in both places it was a 7094. That gave me an opportunity to really understand how the machine worked, and since I was maintaining a very large program, it taught me a lot about program structure. It was a pretty good program as these programs go, and fairly well modularized, although I knew nothing about modularity in those days. But it was non-reentrant code, so when you would call a procedure, they might modify an instruction in the procedure they were calling so that when it got to the end it would go back to the caller without having to have a stack where you branch through something. Of course that was a very error-prone way of doing things. So that was a good lesson for me, to see that.

TVV: So this was led by Tony Oettinger?

BL: Tony Oettinger and Susumu Kuno was a postdoc. Yes. But I was just a programmer, so I wasn't doing research. They'd find an error in the program and they'd just say, "Track this down and fix it." That was my job.

But it was good training. I worry about our current students who may not ever really understand how machines work. Although of course machines are a lot more complicated now than they were then, since in those days they weren't doing speculative execution, they didn't have multiple processors, all that stuff you have to worry about today. But nevertheless, understanding when you're using a higher-level language what the compiler is producing under the covers is really very helpful to understanding what's going on.

I worked at that job and partway through the year I decided to apply to graduate school because it just seemed like it was time. I was learning a tremendous amount, but as I said, I was pretty much self-taught. I thought, "Well, I probably could learn a lot more if I went to graduate school. I'd learn faster." I applied to Harvard and Stanford. It never occurred to me to apply to MIT. And I went to Stanford because I wanted to get back to the Bay Area. I'd been in Boston for a couple years and my family was in San Francisco, so in 1963 I went to Stanford. They didn't have a computer science major then. They had a program. It was between, I guess, math and EE. It was some sort of joint thing.

I went there without any financial support. I didn't even know there was financial support. I wasn't really worried about it anyway because I'd been saving all my money so I had a lot of savings. But my recollection is that on the day I arrived I met John McCarthy[1]. I walked up the steps with him, and I asked him whether he could support me and he said yes. It's highly unlikely that this is what actually happened, so I always think this is an example of how memory is not all that reliable. I think, in retrospect, they probably expected me to be in AI because I had been doing this work on the language translation project even though I knew nothing about AI at the time.

1 Also a Turing Award recipient in 1971

They probably already thought I was going to be working with McCarthy. That's my guess now. It was a very small faculty. I think besides John, there were only Forsythe[2] and Gene Golub. There was a big project in numerical analysis. And Niklaus Wirth came, but he wasn't there yet. So it was pretty small faculty. I don't think there were all that many options about what you would be working on. There weren't many of us in my class either. I think five maybe. I'm not sure. Raj Reddy[3] was in my class.

So I moved back to Stanford in the fall of 1963 and started working with John and taking classes. It was a good thing to do.

TVV: What kind of classes were they?

BL: Whatever they offered. So I took a lot of classes with Dana Scott[4]. He was at Stanford at the time and he was teaching classes in logic. I remember a class in compilers. I remember that we had to write some kind of little compiler-like thing. I don't really remember what it was, but we used to take over the machine at night. It was a B5500 I think. That way we could get fast turnaround, because it was still the days of batch processing, and if you couldn't get hold of the machine, then you would have to submit your program and wait a day or so before you could get your results.

Clearly interactive programming is a big improvement over batch processing, but one advantage of batch processing is that you have to think through your experiment very carefully before you submit it, otherwise you're just wasting a whole day of time. Another advantage of batch processing is that you have to learn how to multiplex your time, because you can't just sit there waiting. [laughs] I think both of those were actually very valuable skills that served me in good stead as time went by.

I don't know what else I took. There was certainly no course in operating systems. I refused to take the course in numerical analysis because I really didn't like that stuff. So I don't know - I took whatever, what people were taking, whatever it was they offered. I remember Jerry Feldman showed up maybe my third year. Probably Niklaus Wirth was teaching the compiler course. That might have been my second year. It's hard to remember.

2 George Forsythe, founder and head of Stanford's Computer Science Department
3 Also a Turing Award recipient in 1994
4 Also a Turing Award recipient in 1996

Meanwhile I was working with John [McCarthy] and I was reading whatever papers were available and so forth. And I figured out, probably in my second year, that I would really rather be in systems. I think I liked that compiler course and I liked that way of thinking. But I decided to stick in AI and try to get my PhD out of the way as expeditiously as possible. I was very interested in what was then machine learning. I was really interested in trying to make machines do planning and stuff like that, but it was a very hard problem.

As you know, that's an area of AI that's changed hugely since those days. Then it was this idea that the program would think like a person and you would try to mimic the way people thought about things. In fact, that was kind of what was going on in my PhD thesis – the *Program to Play Chess Endgames.*[5] There I was thinking about what were the strategies I would use as a person playing that game, and then I built those strategies into my program. But it was limited what you could accomplish with that way of thinking about machine learning. Now that we've switched to the modern techniques, they seem to make a lot more progress.

Anyway, I did my PhD working with John McCarthy. And when I was in my final year, I started looking around for a job, but nobody gave me any guidance. John was not a person who would have done that anyway and I didn't really understand any sort of application process. So I didn't apply to any schools. I sort of waited for people to come to me, which in a way was what was going on with others - I mean Raj Reddy and so forth, because it was the old boys' network in full play at that time.

So what I had offers to do… I must have applied to somewhere, because I don't think these just came out of the blue, but I had an offer at Hayward State – that was because Harry Huskey who was running the department there knew me. I had an offer at SRI – that was because… I can't remember his name right now, but the person running SRI labs knew me. Then I had an offer from Mitre where they knew me, right? [laughs] And I knew that the job at Hayward State would be a very bad idea, because it was a heavy teaching load with very little research, and that didn't seem like a good idea. And I wanted to move back to the Boston area. I actually came and applied to MIT, and I think they would have offered

- 5 Huberman (Liskov), Barbara Jane (1968), *A program to play chess end games, Stanford University Department of Computer Science, Technical Report CS 106, Stanford Artificial Intelligence Project Memo AI-65* me a job as a tech staff probably. Not a research associate. I'm not sure exactly. I decided that probably wasn't a good idea.

So I decided to go to Mitre. That was a good idea because I was also changing fields from AI to systems, and I didn't know too much about systems. I think I'd had that one course in compilers plus my background in computer architecture such as it was. So I had a lot to learn. And so this is now the fall of 1968. I moved to the Boston area. By then I had met my husband. We weren't married yet, but that seemed like a good thing to continue with, and so I moved back to the Boston area and took the job at Mitre.

I started at Mitre in September of 1968, and the first project they handed me was a microprogramming project. In those days, microprogramming was considered to be an interesting research direction. The idea was that they would provide you with a read-only memory and a very simple, small instruction set, and then you could implement a more grandiose instruction set using the read-only memory and this very small instruction set. I had read the paper[6] on the THE operating system and I was very intrigued with the notion of semaphores and I was interested in parallel computing, so I decided to put the project in that direction. I mean, why do a project like that if you're just going to implement a standard instruction set? So I sort of moved it in that direction.

I actually had a chance to meet [Edsger W.] Dijkstra. He came to Mitre. Maybe it was the spring of '69. I'm not really sure. I met with him and we talked about semaphores, and I decided to implement them in the hardware. I had put into the hardware, using the microprogram, sort of a basis for parallel programming. It was a single processor, but this was a way of time-sharing. When that part of the project was finished, then the next part of the project was to use it for something. I built a little multiprogramming system on top of this hardware, taking advantage of what semaphores gave me and some of the other stuff that I've really forgotten about, how you control the interrupt system and stuff like that.

TVV: So this was the Interdata 3…

BL: This would be Interdata 3. Or maybe it was the 4.

6 Dijkstra, E.W., "The structure of the 'THE'--multiprogramming system," *Comm. ACM* 11, 5 (May 1968), 341-346.

BL: Yeah. I think it was… Was it Interdata 3 or 4? I've forgotten. I could look in my… you know. But anyway, it was…

TVV: They were similar.

BL: Yeah. I probably finished the first part of that thing in the first year, and then in the second part of the project, which maybe started in the fall of 1969, by then I was working with one other person, somebody named Bob Curtis who was a tech staff at Mitre. Actually I don't remember that Bob was involved. I'm not sure exactly when we started working together. Anyway, he was definitely involved in the early work on the Venus operating system.

[chuckles] The little computer I developed was called "Venus" and then we developed the Venus operating system. I had also a couple of people working as programmers. I did most of the design and we figured out how to implement this thing. It was an interesting little system. It's been a long time, I really haven't thought about it much since then, but it got me into operating systems and I learned about what was going on, the kinds of abstractions that were being used in operating systems. Semaphores turned out to be handy. That was probably the second year at Mitre – '69 maybe into the fall of 1970.

When that project was finished, Mitre asked me to look at programming methodology. That was a successful project. I'm not sure exactly what these dates are. I actually have some of the tech reports, so I can go back and figure this out.

So that got me into programming methodology. Mitre, as you know, works for the government, and the government puts out request for proposals, and I wasn't in a position yet where I was the person who would be answering those proposals. I was somebody that they were using as a person they could put in charge of a project once they'd brought it in. When I arrived this Interdata 3 project was there waiting for me to work on, and then after that was done, they'd already bid on the "software crisis" request for proposals. And I was finishing up this project, so they asked me to take that on.

That was a marvelous opportunity for me because it opened up a whole new area I didn't know anything about. I started reading the literature, which was not vast, but there definitely was some. The players were the sort of usual players if you think about… I mean there was Tony Hoare[7], Niklaus Wirth[8], Dijkstra[9], and

7 Also a Turing Award recipient in 1980
8 Also a Turing Award recipient in 1984

other people that you would recognize from those days. I guess most of the conferences were in Europe now that I think about it.

TVV: Did you go to the NATO program methodology meeting or whatever it was, the famous one?

BL: I did not, no. I didn't go to many meetings at this time. Later I was in Working Group 2.3, 2.5, the methodology working group. But I never found that kind of format useful for me. I tend to be more "I work by myself." But I read proceedings from these meetings and learned about what was going on, and started thinking about methodology. As I say in my Turing talks, as I was looking at various people – Dave Parnas of course was another big player – and I read all the papers I could get my hands on and thought about their proposals for methodology. At some point during this process, I realized that I had invented a methodology of my own while working on the Venus system, not because I was thinking about it but just because I wanted a way of organizing that software that gave us a sensible way of approaching the software development process.

The idea that I had in Venus was that… I mean to understand the background at this time, you have to understand that there was that ALGOL school of programming which had a good idea and a bad idea. The good idea was that you had blocks, and inner block had private data and the outer block couldn't access it. The bad idea was you had blocks [laughs] and the inner block could freely access all the stuff in the outer block, and so there was a natural tendency to communicate through global variables. That was not such a great idea. Some of Parnas' papers talked a bit about why that was a bad idea, although I would not say that in general it was understood that this was a bad idea.

At any rate, somehow I understood this wasn't a great idea, and I think it has to do with the fact that if you have lots of people working on different pieces of the system and they all can freely communicate through this set of global variables, you're going to have a big mess on your hands. And maybe I saw a mess like that in the Harvard stuff. I'm really not sure where it came from. But I decided we were not going to have shared global variables in developing the Venus system. Instead we were just going to break up the global variable space into what I called "partitions," and each partition would be in the charge of a particular program module, and that module would be the only place you could access that data. Since other parts of the program had to use it, that module would provide procedures that that other parts of the program could call. That's the methodology we used and it worked out extremely well.

9 Also a Turing Award recipient in 1972

Another thing that was going on in Venus, which I usually don't talk about in my Turing talk, is that we were also thinking of the question of "How do you organize a parallel program like an operating system? And how in particular do you control the devices, the shared resources? How do you communicate among the threads that are doing the concurrent processing?" And I was kind of using abstract data types already. I was thinking in terms of the way you do it is the thread makes use of the shared resource, which is actually an object with a bunch of operations, and you call that object and it has some hidden resources which it uses to sort of… it's being shared by all the threads and that's how the control actually happens.

Although I didn't pull that out in the first paper I wrote on programming methodology. I wasn't even really thinking about it until the SOSP[10] History Day where I gave a talk about the history of program structure in operating systems and I looked at the THE system again, I looked at the Venus system, and then there was that Schroeder and Nelson, or Nelson and Birrell - the paper about the two ways of organizing a parallel program, one where you have queues and one where you send messages. I was looking at some of those methodology papers and I realized that Venus was actually one of the ones in the mix there and that there were two structures – one of the shared resources where threads just call operations and everything is taken care of under the covers, and this other structure which is you provide a… it's sort of a CSP-like structure where you have your thread and it's got a bunch of methods that are being called remotely. It's a different model of computation. So Venus had both those things. It was on the shared resource, you just call its operations. It kind of follows naturally from this idea of partitions and just calling operations.

Okay. So I was thinking about methodology and I realized I had a methodology, so I wrote a paper on it and that went into the Fall Joint I think it was in 19-… I think it was probably published in '71 or '72. But meanwhile I also had written [a] paper on Venus, and I submitted that to SOSP. It was the third SOSP. And it was accepted.

By the way, that was the first writing experience I ever had where I had somebody reading my paper and giving me criticism about it. Because John [McCarthy] didn't do that. I had never had that experience. I never had an

10 ACM Symposium on Operating Systems Principles

advisor as a graduate student who was working with me, telling me, "Oh, this doesn't make sense. Reorganize that," and so forth. It was very useful. That was my boss, Judy Clapp. She'd been a tech staff at Mitre even earlier than me, and has very interesting stories to tell. She was serving that role and it was very valuable.

Anyway, the paper on Venus was accepted. SOSP as you know is the top systems conference. It was even in those days, even though it was only the third one, because it was the only act in town. Jerry Saltzer was the… I was one of the prize papers. They've always had this tradition of the top few papers, the award papers, and they go into… they used to go into the *Communications*. Now they go into TOCS.

So Jerry was the chair of my session, and Corby was there, Professor Corbató[11]. And I'm not sure who else was there from MIT. Probably other people from the Multics group. And after my talk I was invited to apply to MIT. So I think I talked to Jerry. He encouraged me to apply, so I did. I also applied to Berkeley, and I could have probably gone to Berkeley, but my husband was not willing to move. We were married by then and he worked for Raytheon and it didn't look like it was easy for him. Certainly not in the Berkeley area. He might have been able to do something down in the peninsula. So I moved to MIT in the fall of 1972. And at the same time, Mike Schroeder was hired, so they hired two people in systems.

I think it was Title IX[12] that sort of opened up things for women. My understanding of what happened was the landscape had changed. All of a sudden there was more pressure on universities to hire women. I don't think all universities were paying much attention to this, but MIT was paying attention. I think it was coming from Jerry Wiesner, who was the president of MIT, because this kind of stuff has to come from the top. Jerry was actually interested in increasing the number of women. The head of the EE department – it wasn't EECS yet, it was just EE – was Louis Smullin, and I think Louis was interested in doing this. Then Bob Fano was the head of computer science, and Bob was definitely interested in doing this. They were looking for a woman and there I was. So it was a mutually beneficial exchange. As soon as I got the offer, I knew I was going to leave Mitre. It was just something I guess I'd always thought would be fun to do, and I decided I was going to do it.

11 Fernando José Corbató, himself a Turing Award recipient in 1990

12 Title IX is a comprehensive federal law that prohibits discrimination on the basis of sex in any federally funded education program or activity.

So I got to MIT. There were only 10 women on the faculty out of a faculty of a thousand. As you know, since you went to MIT, the number of women in the undergraduate population was very, very small. My husband is class of 1960 and there were 16 women in his class. I mean so small, it's really, you know… MIT always admitted women, but they never had very many, partly because they had no housing for them and so they didn't know what to do with them. That changed when… I forget the name of the woman who gave money for a dormitory for women[13], and as soon as they had more space, they started to let more women in. That was in the '70s or maybe the late '60s. I'm not sure exactly when.

TVV: '60s.

BL: It was the '60s? Yeah. So I went to MIT in the fall of 1972 and that was a wonderful move for me. The glory of working as a university professor is that you get to do whatever you want, right? You go to a place like MIT, you have certain responsibilities. MIT is very focused on teaching and quality teaching, and everybody teaches two courses a year, and that takes time and you got to pay attention to that. But as far as research is concerned, you figure out what you're researching on.

Of course there's a downside to this too. First of all, you better have the ideas. You have to figure out what you're going to do. You can do it, but you have to figure out what it is and then you have to raise money. But raising money was pretty easy in those days because those were the days when DARPA[14] was supporting computer science research and it was mostly putting its money into a few institutions. I don't remember whether it was Project MAC still or Lab for Computer Science, but we used to submit one proposal for the whole lab and I just had to write a couple of pages in that proposal to get money. I also wrote NSF proposals just to have sort of another source of money and to practice how to do that. But compared to the situation today, it was a lot easier to fund your research then.

So I came to MIT. The first course I taught was what is currently 6.004. So it's a computer architecture course. This was a weird assignment for me because I had no EE background, and I was actually in an EE department. It wasn't EECS yet. I don't remember when it became EECS. Probably a few a years later. That was a bit of a scramble. Jack Dennis had a graduate student, Clem Leung,

13 McCormick Hall is named after Stanley A. McCormick, the husband of Katherine Dexter McCormick '04, who was the benefactor of the building
14 Defense Advanced Research Projects Agency

who was also a TA for the class, and he helped me. You know, I was keeping sort of one week ahead of the students, learning stuff about circuits and so forth, which really I had no background in. And I definitely had students who were not happy to see me. I didn't feel that I got discrimination from the faculty, but I did feel that the students… there was a few students in the class that were… they'd try to trip me up. They'd try to ask me a question I couldn't answer. Of course I was at a kind of vulnerable position, teaching a course in an area I didn't know. So that was a little bit of a baptism by fire, but I learned how to… I was one week ahead of them, and I learned how to say, "I'll talk about that at another time." [laughs] Meanwhile I started getting the research…

Oh, the other problem was I was in Project MAC or LCS, and they decided I was an AI person and they put me on the AI floor. I think they wanted me to work in AI. But meanwhile I was working programming methodology. Jack Dennis in the spring of 1963 helped me move my office back down to the systems floor, which was a much more congenial place, and that helped a lot. So Jack was very helpful.

So meanwhile in research, I'm sitting there thinking about programming methodology, and I was really interested in programming methodology. I thought it was a very important field. And what I had noticed was that although the papers were very compelling when you read them and they always had an example and you would follow the example and you'd say, "Oh yes, that's very convincing. This is the right way to do things"… And I'm thinking specifically about Parnas' papers because his were about how to structure programming. If you think about the papers at the time, there were papers on "Here's how you design software." So Niklaus Wirth wrote about top-down programming and Dijkstra had that wonderful letter to the *Communications of the ACM* on "Go To Statement Considered Harmful," and really the gist of that paper, if you think behind the scenes of that paper, it was really about Dijkstra's message, which was "We should be reasoning about the correctness of code." The goto was bad because it made it harder to do that reasoning, but the idea that programming is an intellectually difficult problem and that we should approach it in a kind of mathematical way, that was early days and that was a pretty significant step forward to see that way of thinking about things.

But Dave Parnas was writing about modularity, and he was saying, "Here is how we should think about modules." He actually said, "Programs are built of modules, but I don't know what they are." And that was kind of the state of the art at the time. But he had the idea of specifications. He was saying, “Whatever they are, we better describe their connections completely.” So he meant “Whatever their interface is, we better have a complete description.”

I used to feel kind of jealous of the electrical engineers because I thought, “At least they have these components and they connect them by wires, and that forces you to really focus on what those wires are.” Whereas software was so plastic that people used to design without thinking much about those wires, and so then they would end up with this huge mess of interconnections between the pieces of the program, and that was a big problem. That was a problem I was looking at in Venus when I said, “We’re going to have partitions,” but it was just a problem in general. Global variables were a big problem. Just the ability that there was nothing forcing you to do things in any particular way, so you could do whatever you wanted.

So anyway, I was thinking about this. I was thinking about the fact that even though I would read Parnas’ paper and I was convinced that people would read my paper and have the same reaction, you would say, “Yes, that’s a great idea,” but when you start to think about “How do I apply it to my own stuff?” everything fell apart, you just had no idea how to apply it.

I was trying to think about “What can we do to make things better?” and at some point, and I really don’t know when, but probably winter of 1963-64… I mean…

TVV: ’73.

BL: …’72-73, [laughs] I got the idea of data abstraction. And it was this marvelous idea. It came out of nowhere. But once I got it, I could see that this was really going to work because programmers already knew about abstract data types. I mean even if they weren’t thinking about them, because they knew when they used an array in their Fortran program that this not something the hardware had, this was something that you used through a set of operations and under the covers there was an implementation going on. Certainly you knew this in spades if you were using Lisp, which was what I wrote my thesis in, because there you used lists and it was clear there was an implementation underneath and that they were abstract.

So I felt programmers could understand the notion of data abstraction. They already understood about procedure abstraction, and the data abstraction was more powerful because a procedure… Oh, by the way, they were sometimes implementing a data abstraction with a procedure abstraction by having a whole bunch of extra arguments that controlled the different things the procedure was going to do, but that was a mess also. So I felt this was something that programmers would feel an affinity to and something they could understand.

And I think another thing that was important about it was… So it was a bigger module, it fit an idea of modularity – you needed something bigger than a procedure in order to really make progress – and it was also an abstraction. And that was important too because when you design, you need to think abstractly, and having a thing that matches the abstract thought, that helps you with your design. So being able to think in terms of “What data abstraction do I want for this place? What procedure abstraction do I want for there?” this is a design approach. So it was also useful from that perspective.

I was lucky enough to have this wonderful moment in which I had this idea, and as soon as I had that idea, I knew that it was going to go somewhere. So I started working on that idea. And I started working jointly with Steve Zilles. And this was definitely in the spring of 1973.

Steve was a graduate student at MIT and he is an employee of IBM. IBM was in the same Tech Square building that the lab was in. He was older. He was in his early thirties, so he’d been working for IBM for a while and then he went back to school, and he was only I think going to school part-time because he was still working for IBM. And he had had some similar ideas, so Steve and I started to work together. I think Jack Dennis organized a little workshop in the spring of 1963, and maybe that’s how I got connected to Steve. I think Tony Hoare was there, but I could be confused about this. I haven’t tried to figure it out. But Jack was interested in these ideas and he was encouraging me to work on them.

Steve and I started working on this. And having an idea and figuring out what this idea is all about are two different things, so it was just “Here’s a direction to go in.” So we started trying to flesh it out, and we knew that we probably were going to work on programming language as a result because you need to express an idea like that in a programming language so that people have within their grasp the necessary linguistic features to make it all sort of hang together. We were interested in “What would a programming language be like? How could you have type checking that would encompass this notion of data abstraction?” And just the whole thing was a big mystery, but we started working on this.

Of course we read papers, and now I was moving into programming languages from a background where I knew Lisp and Fortran, and of course I’d done reading on other stuff. Steve coming from IBM, which was the big player in programming at the time – they had Fortran, they had PL/I, they had COBOL – so we covered a wide range of programming languages between the two of us and we read a bunch of papers. The one that I found the most… Well, we read the paper on Simula 67, and that didn't quite have data abstraction in it even though it was about classes and subclasses, and you could see how they could be data abstraction. It had no encapsulation, which is a very critical component if you want modularity, and they were mostly interested in inheritance, which we saw as a red herring, so we ignored that.

Jim Morris had a wonderful paper on "Protection in programming languages"[15] I think it was called, where he was talking about modularity and what are the rules that you have to follow in order to get the benefits of modularity. So the benefits of modularity are local reasoning… That's the most important. And Jim said, "Well, a module has some state inside it and then a bunch of code. The first rule is that the code on the outside of the module can't modify that state. And this is clearly essential, because if the code on the outside could modify that state, then I'd never be able to reason about the correctness of that module because all the code in the program would be suspect." But then he said, "But in addition you want modifiability, which means that if I don't like the way that module's implemented, I'd like to be able to replace it with another piece of code implemented in a different way. And so to get modifiability, the code on the outside shouldn't even look at the state. It should only interact through this abstract interface."

So those are in fact the two key pieces of modularity, although in those days and actually even today, another very important component about modularity is that it's a management tool, because it allows you to break up your program into separate pieces. If they follow these rules, then people can work on these pieces independently. In those days, people didn't know what modules were, and there were papers being written that would say things like "A module must not be more than a thousand lines of code." I mean people really did not understand the notion of abstraction, the notion of encapsulation. They mostly only understood that they needed a way of controlling the program development process so that people could be working on separate things.

Anyway, Jim laid out those two principles, and Tony Hoare at the same time was writing papers about… He already had the notion of abstracting from a

15 James H. Morris, Jr., *Communications of the ACM*, Volume 16 Issue 1, Jan. 1973, Pages 15-21

representation to an abstract data type even though sort of they were existing types rather than… So this idea was coming up in very many different places.

But Jim had no idea how to implement this. So then after you say, “Well, these are the rules,” the question is “Well, can I enforce those? In particular, can I build them into the programming language so that the compiler can ensure that you get encapsulation?” So that was what Steve and I were struggling with. How would you implement this? How would you make it work? Jim had a proposal which was basically to use encryption. He talked about “seal” and “unseal.” And that would work. The idea is the module has a key. It encrypts an object when it comes out, it decrypts when it comes in. If somebody’s been mucking around with it, you can tell. That certainly works, but it’s not practical. So we were looking for a… You know, “I don’t want to do it that way. I want something efficient.”

And by the summer of 1973, we had figured out that it was possible to do this with a compiler by having a notion of a linguistic structure that implemented a data abstraction and the compiler would just ensure the abstraction barrier, and the code on the outside would only be able to call the operations. It was nevertheless just a sketch. I mean we didn’t have a language. We just had a proposal for a language. And in that paper, we talked about some issues we didn’t know how to handle. In particular, generics and polymorphism.

Polymorphism was neglected for years. I think it’s relatively easy, when you’re just doing procedures, to ignore the fact that “I don’t want a sort routine that works on an array of integers. I want a sort routine that works in general.” But given the limited range of data types that existed at the time, you can kind of see why people weren’t thinking about that. As soon as you go for data abstraction, you can see that you need some mechanism to allow you to define a data abstraction like a list or a set or a map or something that you just define it once and you don’t have to keep re-implementing it every time you have another type of element. And clearly there was polymorphism in the implementation of the built-in types. So arrays could have floats or ints if you were working in Fortran. So it was there. It just wasn’t sort of pulled out as a mechanism that programmers could get their hands on, and we could see that was going to be an important component of it.

We wrote this paper on abstract data types and it was a big hit. I mean we submitted it to the Conference on Very High Level Languages – which I don’t know when it stopped, it didn't exist for very many years – and it resonated with a lot of people. We finished this paper in the summer of 1973.

Then in the fall of 1973, I started working on what came to be called CLU. So here was this proposal for a programming language with just a few hints of what might be in it and some statements about "It'd be nice if it had polymorphism. It'd be nice if it had exception handling." Exception handling was also… people were trying to figure out what that meant in those days. That was another area in programming languages that people were thinking about but had no real idea of what should be done. So the next step was to sort of really get down to brass tacks and figure out what all this stuff was.

In the fall of 1973, I picked up three graduate students – Russ Atkinson, Larry Snyder… or Alan Snyder, and Craig Schaffert – and they along with me became the designers of CLU. We used to have a weekly group meeting where we would all get together and we'd be working on some particular topic like "What should the exception mechanism be like?" or whatever was the topic of the week. We wrote design notes. In those days, you didn't write them online. You wrote them. My assistant, Ann Rubin, would type them up. We had a very rigorous design process, and we had a group meeting that was attended by quite a few more people than just us. So Steve was coming, Jack Dennis used to come. Other people. Students of Jack's. So we had a pretty big group and we would just hammer out… Everything we looked at we'd try to look at from all possible directions and figure out "Of these two approaches, what's the benefits? What are the disadvantages?"

We had a very rational design process, and that's how we got CLU together. And the students implemented, and we implemented as we went. We started off with a compiler written in a language called MDL – M-D-L – which was a Lisp variant, and a very small subset of CLU, and then we bootstrapped. You know, we used that to implement to CLU. And of course CLU itself was a big test case for the methodology, because a compiler's a pretty big program, and so if you can build a compiler, especially when you're working in sequential programming, that's a good test case. So it was very useful for us to be implementing our own compiler since it was forcing us to make sure that the linguistic mechanisms we were providing were powerful enough for the compiler.

So we're implementing CLU, we're designing CLU. We'd figured out… We call these things "clusters." I think the word "cluster" was even used in that paper with Steve. We couldn't think of a good name. Eventually we just used CLU, C-L-U, the first three letters of "cluster." And I guess that… I'm not going to talk about the actual features of CLU, but I do want to talk about it in more general terms.

CLU was way ahead of its time. It wasn't just that it had data abstraction and nobody else had this. The only other project that was going on at the time that was looking at data abstraction was Bill Wulf and Mary Shaw were working on a language called Alphard at CMU. So they were also looking at data abstraction. A big difference between them and us was that I came from Lisp, I believed in the heap. They were very much in the ALGOL world or the… There were a lot of arguments in those days about "Pointers are bad." So they wanted everything to be on the stack. Of course you can avoid garbage collection, but it made everything much more complicated for them. So I think it slowed them down. Whereas I was coming from the Lisp camp. I believed in garbage collection, I believed in the heap. I didn't think pointers were bad provided you could get your type checking sorted out. I was however very in favor of strong type checking, and as I say in my talk, this is partly a reaction to Lisp, because I found it so annoying that they didn't do static checking and you can save an awful lot of time in the program development process if you have a good compiler doing static analysis.

Lisp had another thing that influenced me a lot, which was separate compilation. That was also very important. Right from day one, CLU was always separately compiled. In fact, our idea was you would put into the program library a description of the interface of a module first, and then you could compile code that used the module even though the module wasn't implemented yet. And if you wanted to, you could provide a sort of simpleminded simulation of the implementation as your first implementation if you wanted to go further. So we were carrying this notion of separate compilation, - we were pushing it as far as we thought we could.

CLU had data abstraction. We knew we needed polymorphism. And polymorphism was a challenge for us because of the fact that when you write a data type or a procedure that's parameterized by some arbitrary type T, sometimes not every arbitrary type is a legitimate parameter. So if you're talking about a sorted set of T, then you have to have some way of comparing the T elements. And not every type has an ordering on it. And we didn't know how to… And we wanted to capture this statically. We didn't want this to be some sort of dynamic thing where we discovered at runtime, "Oh, the operation we need is missing." We wanted to ensure at compile time that there was such an operation.

Finally we invented what we called "where clauses," where we would simply list the set of operations with their signatures that the type was required to have, and then the compiler could check when it was compiling a use that the parameter type being provided had the operations that were required. Of course we captured only syntax, not semantics. You know, we said, "It has to have an operation named less than…" With two Ts returning a Boolean, we didn't say it was an ordering relation. So that would have been part of its specification. You would have had to reason about this outside. But that's about as far as you can go with a compiler, because a compiler doesn't reason. You know, it can do simple parts of the reasoning but not the full reasoning.

Interestingly there's something called type classes in Haskell and these are strongly related to where clauses. They are pulling the requirements for a polymorphic module, saying, "Here's a set of operations, here are their signatures," and then you can put a specification with it. But CLU had these in there as where clauses. So that was our solution for polymorphism.

We had an exception handling mechanism. We thought exceptions are very important, because from programming methodology point of view, you would like the specifications of your operations to be complete if possible. Not partial but complete. So covering the entire range of possible inputs. Since if you ever have "anything but true" as the precondition for your call, there's a potential source of errors there because somebody using your module forgets, whereas if it's covering all the bases, then you can be certain that those errors are not possible. But when you try to make a procedure total, then you have this problem that you can't return the same way over the entire space of inputs. So you need some way, we thought, of bringing this to the attention of the caller.

The way that people manage this problem today and even then when they don't have an exception mechanism or they think it's too expensive to use it, is they play a game. So they'll say, "Well, you return a value and a special piece of this value tells you what's going on." Like "I return a pointer to an object, but if the pointer is null, this means something." And the problem is that's very error prone. "I'll return an integer if the integer is negative." This has a special meaning but people forget to test. In some sense, it's the same as having a partial spec. It's slightly better, but it's also very error prone. So we wanted a mechanism that told the user, really push those results into another part of your program.

This all seems so commonplace today because this is how Java's mechanism works, but in those days, it didn't exist. So we invented that stuff. And we worried about a lot of the problems with exception handling. One of the problems with checked exceptions in Java is that people don't like to have to write the code to handle exceptions that aren't going to happen in their program. "I just checked that my index is in bounds, so why do I have to write a catch clause when I call the lookup? Because I know it's not going to happen." So we handled that by turning those into a special failure exception.

So CLU had an exception mechanism. That was another large part of our design, working out all the details of how that would work.

And then the third part of our design was iterators. That was one we didn't foresee going into the project. The other two, I think they were in that original paper, but iterators was not something on my map. But we had come to realize we needed an iteration mechanism because many data abstractions are collections, like sets and maps and stuff like that, and when you collect, it's usually because you want to do something with the collection, which is often iterating over it. Although you can figure out ways of doing iterations in the absence of a mechanism, it seemed more elegant to have a mechanism. And as I have told the story many times, we went to Carnegie Mellon, we visited with Bill Wulf and Mary Shaw and their group. They told us about something called generators, which actually was coming out of AI ideas. So we listened to this and generators were kind of like what iterators are in Java today. They were objects with a bunch of operations to get the iteration started and so forth.

So we could see this was a nice solution, but we thought it was kind of inelegant and overkill. And so on the plane going back to Boston, my student Russ Atkinson invented iterators, and iterators are tailored for use with a for loop. You call the iterator to start the loop. Every time it's got a new value to provide you, it uses a special return instruction called yield. We then run the loop body. At the end of the loop body, we return back into the iterator exactly where it yielded so it just continues in its control flow. When it has nothing more to yield, it returns, and that terminates the loop.

It's a limited form of coroutine, because you don't have the ability to sort of keep them running, multiple of them or so forth. For example, you can't run over two trees and check for the same fringe using iterators. They have to be nested. But we decided – and this is kind of the 90% rule for programming language design – most of the time, this somewhat limited use of iterators was what you wanted. So rather than have a more complicated mechanism that got you further but wasn't as convenient to use in the normal case, we would go for the simpler mechanism. And it was nice too because it had a very efficient implementation where we simply passed the loop body as sort of a hidden routine to the iterator, which called every time it yielded. So very straightforward.

So that was CLU. And by 1978, we had a compiler that was working well, we had a language, we had a reference manual, we had users. It was being used at over 200 sites at one point. It was being used for building big software.

I should say that it was important to design a language for several reasons. One was people have to write programs in the language for you to understand whether you have the right mechanisms in place. Our users, if they didn't like something, they would complain and we would think about whether our mechanisms were powerful enough.

Another is performance matters. So you need to think about "How expensive is it?" For example, our exception mechanism, it only cost about twice as much to signal an exception as it did to do a normal return. If exception mechanisms are expensive, which is unfortunately the way things are in modern languages, people don't want to use them. Even though they might be wrong. I mean it may be that the exception case doesn't happen very often, and so if you look at the overall performance of the program, the cost of the exception doesn't matter very much, maybe. But we felt it was important to have an efficient exception mechanism so that that barrier to use would not exist.

Then you want a mathematical definition. You want a real mathematical object that people can understand the meaning of. So that was another reason why it was important to do programming languages. Then we wanted a language because people think in terms of… Programmers think in terms of programming languages, so seeing those features just sets the stage for figuring out what to do.

And by the way, once the features exist, now you can start to simulate them in other languages and people will still see it's a data abstraction. So for many years 6.001, our introductory course that Gerry Sussman developed, they were teaching data abstraction, but they were using basically a record of pointers to (provide) the methods of the data type. And after data abstraction exists, you can kind of see that's the way to go. So that's fine.

That's really the story of CLU. And it got to be 1977 or 1978 and CLU was pretty much finished, and I started to think about what to do next. At that time, I had already started teaching 6.170.

TVV: Which is what?

BL: 6.170 for many years was the second programming course in our curriculum. And I was asked to develop this course by Corby and other people in the department who thought we needed such a course. So our students would start with 6.001, which was taught in Lisp, or Scheme actually, and they learned how to build little programs. And the idea in 6.170 was "Okay. Now how do you build good, big programs?"

So I developed this course. It was all based on… It was about programming methodology, how do you do design, how do you use data abstraction, how do you do modular design. It was really in line with my interests, and I taught it for about – let me think, '77 – probably 20-25 years, something like that. I to this day still get people telling me how important it was for them, what an impact it had on their career, because it really did teach the students how to think about modular design and how to organize a big project.

And we still teach it. It's just morphed into another course, which… It was too much work for the students. This is an MIT problem in general – courses are too much work for the students. So they sort of tried to divide it. I'm not sure this has been terribly successful. I have a feeling these courses tend to… more and more material accretes in the course as you go by.

Anyway, so I was thinking about what to do next. I could have continued working on programming language stuff, but I didn't feel like I had any great ideas. I didn't see another abstraction mechanism. I didn't see a way where I would be able to make a big impact. I mean CLU, this work had made a huge impact, so I'm sort of looking for impact like that. And iterators were really important too. But parallel computing, I didn't have any great ideas about what to do about that. So I thought, "Well, no." [laughs]

I also thought about commercialization, but I decided that… In these days maybe you can commercialize by putting stuff out on the web and people will start to pick it up and maybe there will be users who contribute to the implementation and so forth. I think it's still an awful lot of work. I think the people who put it out there end up spending most of their time focused on that. So it's not really research. It's much more development. So that didn't seem like a very good direction to go in.

I was looking around thinking about what to do, and I started to think about the ARPANET. And I really don't know, I don't remember what it was that caused me to see this problem lurking in the ARPANET. It wasn't a problem that I invented. Bob Kahn[16] had been writing papers about the ARPANET. So in those days people did email, people did FTP, people did remote login. That was kind of what you did on the Internet, and I was using email already. I mean people have told me email didn't exist so early, but it existed in the '70s because I was using it. But there was a dream of writing distributed programs where they would have components running on different computers and they would communicate by sending messages, and nobody knew how to do that. So I thought, "Ah, this is a great problem," and so I jumped into distributed computing.

This was in the late '70s. I just switched directions. I didn't switch totally because I was still working on 6.170. I had been thinking about "How do you reason about the correctness of abstract data types? How do you write specifications for abstract data types?" A lot of this was being taught to the students in my course. I was also…

TVV: If I could, I'm going to pause you for a moment.

[Recording was paused for everyone to take a break]

BL: Okay. So where were…? [laughs]

TVV: [A lot of it was being tied to your students…]

BL: In 6.170, yeah. And I was working with John Guttag. I forget when John came to MIT. He had done his thesis on specifications of abstract data types. Steve

16 Robert ("Bob") Elliot Kahn, himself a Turing Award recipient in 2004

Zilles had done a similar… didn't quite finish his thesis, but a similar kind of research. So John and I were working together. We started working together on 6.170. We wrote a book. I was still interested in the programming methodology stuff. Not so much the programming language stuff, but the programming methodology stuff. But I jumped into research in distributed computing, and I really stayed in that area after that, with a few diversions into other stuff, and had a good time.

I thought it was ironic in a way that I decided to not look at concurrency when I was working on CLU on the grounds that I had enough on my plate and that would have just been a huge distraction. When I got into distributed computing, of course concurrency came right back, so I'm thinking about concurrency again since clearly you have all these computers and they're all working in parallel and so forth. In a way, distributed computing is a great place to think in terms of abstract data types because you want to have different objects running on different machines, they're going to communicate.

One of the things I was thinking about in the early days of the first project, which was the Argus project to develop a language to implement these distributed programs, was "What is the communication mechanism?" I ended up strongly on the side of remote procedure call. You know, that on my remote machine I have an object, it provides operations, and over here I call those operations. And then under the cover, stuff is passing. Argus was one language system, so we would have been able to… If you're running on one machine and you have one language, you can do a much more efficient remote procedure call than you can do if you worry about heterogeneous machines, different programming languages, and so forth.

Anyway, I started working in distributed computing and that was a long haul. In the '80s and the '90s I had some great students. I'm not sure what to talk about there. It's…

TVV: Well, let's see. Who influenced you? Were you… How about Lampson[17]? Was he a… his stuff at Xerox? Or…

BL: Well, so I was definitely at this point going to operating system conferences. Certainly I… Another course I taught at MIT was 6.033, the systems course. I taught that many times. And so Multics and various operating systems, all that stuff.

17 Butler W Lampson, himself a Turing Award Recipient in 1992

I wouldn't say… You know, it's hard to answer. I wouldn't say that Butler's work particularly influenced me. There was a group in the '70s of DARPA contractors who got together a couple times a year to talk about programming languages and programming methodology. So Butler was in that group, Bill Wulf and Mary Shaw were in that group, I was in that group, Jim Horning, the Euclid developers. So there, I used to go that meeting every six months or so, and there was a lot of exchange of ideas. You look at a language like Euclid and you can see data abstraction, specifications, all that stuff was going on.

But when I moved into… No, the thing that influenced me was transactions.

TVV: Right.

BL: Yeah, that's right.

BL: And that was coming from Jim Gray, from System R, from the database community.

TVV: Right. And then Lampson and Sturgis with the stable…

BL: The stable storage[18], but that was really much more a 6.033 topic than it was an Argus topic. So what we did in Argus was we brought transactions into the programming language. We were interested in the point that when you make one of these remote procedure calls, you can't be sure you're going to get an answer because you're talking to a different machine and there could be a reason why communication isn't working, and you will never know what that reason is because you might just not have been waiting long enough for the answer to come back or it might really be down. Right? This is the beauty and the horribleness of distributed computing. I think it's kind of neat myself, that you just have to get in this mindset where the lack of an answer tells you nothing, right?

The problem is here I am on the calling side and I don't want to wait forever, so what do I do? Well, what we thought you did was you're running a little subaction and you abort it, and that means even if it happened over there, it hasn't really happened and so you don't have to worry about it. You could try an alternative technique and so forth.

That was a big piece of originality in Argus. We ran the whole thing as transactions. So we had objects that were instances of data types. They ran on

18 Lampson, Butler W. and Howard E. Sturgis, *Crash recovery in a distributed data storage system*, Unpublished technical report, Xerox Palo Alto Research Center, June, 1979, 25 pp.

individual machines. And then we ran computations as transactions, and every time we made a remote call, we ran it as a subaction. That was sort of the position of Argus. I don't think it was necessarily a good idea because it was complicated and expensive. I'm not sure I would do things the same way were I to do it today. I would probably use a much simpler model of computation.

One thing that's interesting, a piece of history about Argus though, is that X-Windows came out of Argus. Bob Scheifler was working for me. He was one of two staff members who were big implementers for us. He's a marvelous implementer. We needed a way of debugging distributed programs you're running, and he came up with X-Windows because what was nice was you could have a window over here watching that… we called them "guardians," these objects, and another one over here watching this guardian. So it gave you a very nice debugging environment.

Then Jerry Saltzer had been in charge of Project Athena and Kerberos had come out of that. That was a big hit because it was public domain, and so Mike Dertouzos thought, "Well, let's try to make X into the public domain," so we formed the X Consortium. This was kind of the start of windows being the way that you managed your system in a distributed world. It wasn't the first windows. There was something called W I think which preceded it. W, X – "X" is after "W." But it was just an interesting little sidelight on what was going on. It wasn't my invention, it was Bob's.

So we implemented Argus. I mean at this point, we're trying to make some sense out of "What are distributed systems?" and there's a lot of work in the operating system community, people are thinking about "Maybe I just have a great big heap, and programs run and they share these objects in the heap." I'm not sure that this work ever really went anywhere in the sense of "People are building programs using that," because what happened was the big RPC[19] model came in. The idea was you build your components, they communicate through an interface that's described in the library, and software connects them together, so you get heterogeneity. The performance is not great, but that was the idea.

Anyway, I worked on Argus and then students who were working for me at the time, we were thinking about data abstraction and concurrency. So Bill Weihl wrote a thesis on commutativity and how you can use the specification of an abstract data type to figure out how much concurrency's allowed. In a database

19 Remote Procedure Call

at that time, they used two-phase locking. That's a mechanism that has no understanding of any meaning. It's just a technique that you can use that will guarantee serializability. There was also optimistic concurrency control – not used so much then, but used a lot later. But Bill's idea was "If I understand the semantics of the operations, I can get more concurrency than would be allowed by these concurrency control mechanisms that don't understand the semantics." So that was an early piece of work that came out of that group.

Then another thing that happened in the '80s was we invented what's essentially Paxos. We invented something called "viewstamped replication" – this was another one of my students, Brian Oki – because we were interested in reliability and availability. I mean I thought of distributed computing as being both a blessing and a curse. If your machine went down in a non-distributed system, you can't do anything until it comes up. With a distributed system, you could have stuff someplace else so maybe you could continue working.

On the other hand, if you have stuff someplace else and you don't have a way of controlling things, then there's more than one failure that can cause you to stop working, so what do you do about that? We came up with a replication technique to ensure that everything worked properly, and as long as f out of $2f+1$ nodes were working… So yeah.

TVV: Okay. So why don't you tell us about the Liskov substitution principle?

BL: That was an interesting thing that happened in the '80s. At the same time that I was developing CLU and Bill and Mary were working on Alphard, Alan Kay[20] and Adele Goldberg were working on Smalltalk. On the west coast. Although it may seem a little strange these days, in those days it was a long way from the east coast to the west coast, and of course we had no conference calls in those days too. That whole business about object-oriented programming was developing on the west coast and on the east coast we were mostly working on data abstraction, and the two worlds were kind of separated. So I knew the name, but we didn't run into each other at conferences and there wasn't much crosstalk going on.

In the 1980s, I was asked to give a keynote at OOPSLA[21], which I think it was maybe the second OOPSLA. It hadn't been in existence very long. So I decided that this was a good opportunity to learn about what was going on in object-oriented languages. So I started reading all the papers and I discovered that

20 Himself a Turing Award recipient in 2003
21 Object-Oriented Programming, Systems, Languages & Applications is an annual ACM research conference

hierarchy was being used for two different purposes. One was simply inheritance. So I have a class, it implements something, I can build a subclass, I can borrow all that implementation, change it however I want, add a few extra methods, change the representation. Whatever I want to do, I just sort of borrow the code and keep working on it.

The other way it was being used was for type hierarchy. So the idea was that the superclass would define a supertype, and then a subclass would extend this to become a subtype. I thought this idea of type hierarchy was very interesting, but I also felt that they didn't understand it very well. I remember reading papers in which it was clear they were very confused about it, because one in particular that I remember said that a stack and a queue were both subtypes of one another. This is clearly not true because if you wrote a program that expected a stack and you got a queue instead, you would be very surprised by its behavior. The difference between LIFO and FIFO is a big deal.

This led me to start thinking about "What does it really mean to have a supertype and subtype?" And I came up with a rule, an informal rule which I presented in my keynote at OOPSLA which simply said that a subtype should behave like a supertype as far as you can tell by using the supertype methods. So it wasn't that it couldn't behave differently. It's just that as long as you limited your interaction with its objects to the supertype methods, you would get the behavior you expected.

This was an informal definition just given based on intuition. It's intuitively right in some sense. You can see how you understand the supertype, you write some code in terms of the supertype, whatever object you get should behave that way you expect. Otherwise how can you do this independent reasoning about behavior?

Later on Jeannette Wing, who actually had been my master's student I think and then John Guttag's PhD student, approached me and said, "Why don't we try to figure out precisely what this means?" So we worked together on this in some papers that got published a bit later.

Meanwhile I was working on distributed computing, I was particularly interested in viewstamped replication and some of the other work that was going on in my group at the time, and I wasn't really thinking about this until sometime in the '90s when I got an email from someone who said, "Can you tell me if this is the correct meaning of the Liskov substitution principle?" So that was the first time I had any idea [laughs] that there was such a thing, that this name had developed. Technically it's called "behavioral subtyping." You know, it says subtypes behave like supertypes. So I just thought that was very amusing. I discovered there were lots and lots of people on the Internet having arguments about what the Liskov substitution principle meant. So it was nice to have something that had an impact like that. I would say you put data abstraction together with type hierarchy and now you have sort of modern object-oriented programming.

But that was a little deviation from the work I was doing, which was all distributed computing. I worked on distributed computing through the '80s and '90s. And it's kind of hard to remember all the different projects that were going on. Sometime in the fairly early '90s I started working on the Thor project, which in some sense was the opposite of Argus. So Argus was a project in which the objects were the components of the distributed system. Thor was a project in which you had a client-server model, the objects were stored in the servers, and the clients interacted with one another only through their use of the objects.

So it was much more of a database view of the world than Argus was. And we were still running transactions, but now these were transactions… they weren't distributed transactions as in Argus. They were one-machine transactions where a client would run against the objects at the servers and at the end either commit or abort, and that would cause the global state to change. And I think that might be more productive way of building applications. That was truly an object-oriented system as opposed to a database system. We used a very interesting form of optimistic concurrency control and we did a lot of work on cache management and other techniques that made the system perform well.

As you know, performance is greatly overrated in the minds of computer scientists. On the other hand, performance matters a lot when you're building a platform that you would like to people to build applications on top of. So it matters a lot in an operating system. It would matter a lot in a system like Thor or Argus because you would expect to build on top, and any problem with the performance that exists at the level of the implementation of the platform will be multiplied when you get to the top level.

In the ’90s, in addition to the work on Thor, my group did two other things that I thought were notable. The first was Byzantine fault tolerance and the other one was decentralized information flow control. The first one… I’m not sure exactly the order these happened. They’re probably simultaneous. The first one happened with my student Miguel Castro. What happened was I saw a request for proposal from DARPA talking about malicious intruders on the Internet and what can we do to counteract their impact. And I gave this to Miguel. I said, “Think about this. See if you can think of something interesting we might propose to do.” And he thought, “Well, maybe we should look at this question of replication in the presence of Byzantine attacks,” and this ultimately turned into that work on Byzantine fault tolerance.

It wasn’t that people hadn’t worked on it before, but they had been mostly theoreticians, and so they weren’t thinking about a practical technique, one that would have a low cost or as low cost as you could manage, and it would really be able to be used in practice.

TVV: So we should say what “Byzantine” means.

BL: Oh. “Byzantine” means arbitrary failures. The work I did in the ’80s on viewstamped replication, otherwise known as Paxos, assumed benign failures where a computer was either running or it wasn’t. You know, a message either arrived or it didn’t. It wasn’t garbled in some way. The computer either failed by crashing or it was running correctly. You know, the message arrived intact or it didn’t come at all. Or maybe it arrived and it wasn’t intact, but you recognized it right away and you could throw it away.

With Byzantine failures, the computer keeps running. It’s not running properly anymore, but it’s running nevertheless. Of course mostly this will happen and you’ll be able to know that it’s not running properly, because it will be doing weird things, but a real Byzantine failure is one where it continues to run and it looks like it’s okay. And there were examples of this happening. For example, you’re probably aware of the stuff that was going on in the networking community where a little flip of a bit in a message caused all sorts of problems to develop in routing and so forth of the message.

In the case of Byzantine failures in running a computer, these were mostly the result of attacks. And it’s interesting, when I first started working in the Internet in the ’80s, we were just a group of pals. Everybody was a friend. It started off as a group of universities connected by the ARPANET, and we didn’t really worry about attacks because nobody was interested in doing attacks. We were just interested in “How do you make things work?” As we moved into the ’90s, this was increasingly not a valid assumption. Once the World Wide Web came along…

And I should say right now that, like everybody else I know in the sort of mainstream computer science community, I didn’t see it coming. I had no idea that we were going to move from these computers that my pals were using to something where the whole world started to use it. That was an amazing transformation.

Anyway, as it became this thing that the whole wide world was using, then the problem of bad actors who would try to launch attacks on people’s computers for purposes we know today of really bad stuff, like encrypting your files and then demanding blackmail money to give you the key and stuff like that.

So in the ’90s, we were in a transitional period where we began to think that maybe these attacks actually mattered. So the question of how to do a replication technique that would handle Byzantine attacks where one of the replicas was in fact behaving maliciously and would appear to be behaving properly but was actually not behaving properly. For example you’d say, “Run this operation for me,” and it would come back and say, “Okay, and here’s your answer,” but in reality it had done something entirely different. That was an example of a Byzantine attack [really “failure”].

So Miguel and I worked on this problem, how to do a practical protocol. For a benign failure you need $2f + 1$ replicas. So to handle one bad replica, you need 3. For Byzantine you need $3f + 1$. So you would need four. And you also need a more complicated protocol. And I’m convinced that one of the reasons we were able to figure out how to do this was because we had done viewstamped replication in my group earlier, and looking back at what happened, I see Byzantine fault tolerance as being viewstamped replication extended a little bit. It has an extra phase in the protocol, it has an extra replica, but it’s quite closely related. So I think it helped Miguel and me when we were trying to figure out how to make this work that we had that background to depend on.

The other major deviation was decentralized information flow control. This was work I did with my student Andrew Myers. For security in systems, there had always been two different approaches. Access control, which controlled what a program could access or what a user was allowed to access, but once something could be accessed, you could do anything you wanted to with it. Information flow control didn't control access. It controlled what could happen with the information after the fact. So if you were entitled to read secret files – and this came out of this kind of work in the military – then you would not be allowed to expose that information except to something where secret information could go. So it wasn't the control of what you looked at that mattered, it was the control of what you did with it.

These systems had always been based on a centralized notion of who was in control of things. There was Top Secret, there was Secret, there was Confidential. Somebody was classifying everything and then the system just followed the rules based on those sort of centralized notions. What I did with Andrew was to decentralize this so that individual users could put labels on their data based on their own desires for the control and then make the whole thing work in the same way, controlling the access. And that's led to quite a bit of work in programming languages and operating systems after the fact. We worked on Thor and we worked on these two interesting directions, and we did some programming language work too, especially with Andrew who was a programming language person working in my group.

That took us through the end of the '90s. Then I took a couple years off and worked for a startup just to see what it was like. This was a startup called SightPath that was fairly soon after I joined bought by Cisco, and I ended up working there for two years. I would say working in a company was not my cup of tea, so I returned to MIT and I became head of the computer science part of our department. I continued working on distributed computing, developed a system called Aeolus, which was a decentralized information flow control system with a programming language component. Then I started working in the administration at MIT. I was Associate Provost for Faculty Equity. So that was what was happening in the 2000s.

Then I decided to retire, and I retired a couple years ago. But in 2012 I sort of stopped working in distributed computing and since then I've been working in programming languages and multicore machines just to sort of continue the… I like to move around and I decided it was time to move around for a while, so I've been doing that.

So that's sort of what I've done.

TVV: Let's see. A couple things. When you were doing the Provost thing, what was that and how did it work out?

BL: Well, when I was Associate Head for Computer Science, I was in that position for three years and I hired five women in three years after a long period in which we had somehow or other never been able to find women that could be hired. And these women have all done extremely well, so it wasn't like I was making compromises to hire them. I just managed to find them where other people couldn't see them.

So the person who… It's a long, complicated story. But anyway, it was this kind of thing – sort of trying to make sure that departments all across the Institute were doing the right thing as far as women and underrepresented minorities were concerned. Mostly it had to do with first of all you want the data. So you want to know what's happening. Then you… There's a lot of… You want to make sure the search is carried out properly and you want to make sure that department heads understand certain basic things. Like for example it's their responsibility to make sure that their junior faculty are not being asked to do the wrong jobs. For example a lot of people don't understand that women are much more willing to say yes to things than men are. So if you say, "Would you teach this lab course? And oh, by the way, it's really a hard job," a young woman assistant professor is much more likely to say yes than a man is, so you have to sort of take this kind of stuff into account. So it was just that kind of stuff. Trying to make the Institute better by making sure that we were doing a good job of this.

TVV: And you think it's made a difference?

BL: I think that it helped when I was in the position. I think this is the kind of stuff that you have to pay attention to, it has to come from top, and if you stop paying attention to it, I think things will slide. So I haven't… I did my bit and now I've passed it on to others.

TVV: So in general, how do you feel about MIT?

BL: Oh, I absolutely love MIT. I think it's been a great place to work. I mean I would say the lab is also a great place. MIT has this peculiar sort of matrix organization where there are departments and then we do our research in a lab. So I've been in… it was Project MAC, then it was Lab for Computer Science, then we merged with the AI lab. Now it's CSAIL – Computer Science and AI Lab.

But MIT has been wonderful – the quality of the students, the quality of the faculty, the interest in doing research and really understanding things from first principles. I mean I find this is true even in our undergraduates. They want to really understand things and so it just makes a wonderful environment. I also have found my colleagues very collaborative. I love to come to work every day. It's just been great.

TVV: Let's see. A couple of other things that I wanted to make sure we covered. Tell us a little bit about the Turing itself. When did you hear you were getting the award? What's it been like?

BL: I received the award… It's the 2008 Turing Award. I received it in 2009. This has been a mystery to me why they have this offset in years. It's something historical.

I learned about it because I got a phone call from Brian Randell. He was the head of the Turing committee that year and I knew him from the old times. He was another one of the people doing work in programming methodology in the days when I was in that field. That was a huge surprise. He says to me, I picked up the phone, "This is Brian Randell." I hadn't seen him in years. He said, "You better sit down." [laughs] So that was great.

It's wonderful to receive the Turing Award because it's such a validation of what you did. And what was interesting for me was that, because of the way I have moved around, I had stopped working in data abstraction, programming languages, methodology. I'd been just working in distributed computing and I wasn't even thinking about that stuff. So I got the award and it caused me to go back and think about what things were like in the old days.

The first surprise was that I discovered my students didn't really know there was a life before data abstraction. And they actually didn't even know some of these old papers. I was pretty surprised by that. So I went back and I reread all those old papers. It was very interesting to look back. And these are extremely good papers too. It's really a part of our history that should not be forgotten.

As I like to tell people, when I got the award, there's a lot of buzz on the Internet, and my husband was online every day looking at what the stuff was. And of course not everything you see out there is nice. In fact, [laughs] many things are not so nice. So there's nice things, there's not so nice things. But anyway, one thing he found out there one day was a quote that was roughly "What did she get the award for? Everybody knows this anyway." And I just thought that was the most amazing compliment, though I don't think it was intended that way. But to realize that these early ideas, not just mine but of course all the work that had gone on in those early years to understand data abstraction and specifications and programming language and so forth, to understand that they had moved so into the mainstream that everybody knew them and they were the basis of how you wrote programs, I mean that was just a remarkable thing to understand.

TVV: Do you want to say a little about Java, which certainly shows a lot of toolmarks from your…

BL: Well, so Java, I was very glad to see Java come along because it was the first mainstream programming language that really had these ideas in it. It really did have data abstraction and object-oriented programming. There really was a notion that superclasses ought to be supertypes, the notion of interfaces define a kind of behavior. They enforce the Liskov substitution principle in the sense that a subtype, a subclass has to provide the methods with the right signatures the superclass has.

I would have done it differently had it been me, and Andrew and I, Andrew Myers and I did try to convince them to put polymorphism into the language. When it first came out, it wasn't there. Of course when you design a language like this, you have to decide what's important to concentrate on, what you're going to leave for later. So it's not that this is necessarily the wrong thing. It's just not what I would have done. I think that it sort of opened the doors for this to become… In a way, the reason it is mainstream is because it's there now in the languages that people use on a day-to-day basis.

And it's moved into C++ in C++'s kind of form. I wish people would enforce encapsulation better. I think they do a better job in C#. Sometimes you have to violate encapsulation, like when you're building a platform, like a debugging platform. But it's better if you could limit that to people who are entitled to do it rather than having it be something any old programmer can do.

You know, there was a lot of stuff, sort of wisdom in the old days that people maybe lost. So one thing we learned was that reading code mattered much more than writing code, because code is written once but read many times, first by the author, then by others. There was another one I was going to tell you about. Well, it will come to me. Anyway, I feel like some of these lessons from the past have been forgotten, and maybe it's just as well because it means we've made progress.

TVV: Well, another thing that we should talk about to get on tape is the people that you've learned from and that have influenced you.

BL: Well, I worked with John McCarthy. I would not say that John was a very hands-on advisor, but I also felt that John was a fair-minded person and I certainly never felt any sort of negative "You're a woman, you can't do it" or stuff like that. So he was good and he handed me my thesis topic when I couldn't make progress on machine learning, this "Program to Play Chess Endgames," which he thought I would be good to work on because I didn't play chess. And he wanted me to approach it from the point of view of "I'm just somebody learning and I see what the books are saying and I think about that from a heuristic point of view," which turned out to be quite effective.

I think that Niklaus Wirth was also important. I knew him as a student. He tried to convince me to switch into programming languages and compilers, and he was right that it was really much more my field. I didn't do it because I felt I would get finished with the PhD faster by sticking with AI, which I think was also correct.

Of course I've learned a lot from lots of people whose papers I've read and whom I've talked to, technically. I mean Jerry Saltzer, all those years I was teaching 6.033 which was his course, our systems course, and his way of thinking about systems. Corby, another one. Bob Fano, another one. Bob Fano is a wonderful person. He's the one who hired me. He told me that I was an engineer. I hadn't actually realized that, and he was so right. [laughs] I think I didn't know what an engineer was. And then Jack Dennis was a big help. Jack was the one who got me off of the AI floor down onto the systems floor and in general was just encouraging and helpful, especially in those early years when I was finding my way. So I would say those are the people that I think back on and I feel were helpful.

TVV: And let's see. Oh yeah. So how has this experience been for your family?

BL: Well, I do have a family, so I did get married. So you can have a career and a family. That is a message I always try to convey to young women. I have a son. I have a granddaughter. I had my son before I was tenured. I felt that "I'll figure out a way to make this work." I think that in fact is a very good way to run your life. Rather than to think through all the things that might go wrong and worry about it in advance, you just figure you'll make it work somehow. I don't know. My son's a computer scientist. [laughs] Though he's more on the theoretical side than I am.

TVV: He's a professor, right?

BL: No, he's not. He was at William and Mary for a while but now he works for Mitre and he does security research.

TVV: Cool.

BL: Yeah, great. And my husband has been very supportive and helpful. I don't think I could do it without… You need a helpful spouse. So I think it's… I don't know how it's been for my family, but I do have a family and it's all worked out okay.

TVV: Well, let's see. You've won a lot of other awards.

BL: Yeah. Well, I won the von Neumann Medal from the IEEE actually a few years before I got the Turing, and I've also got some honorary… You know, I've got… Yeah. I think there's sort of a tendency once you get one award, sort of more come along.

TVV: You've got a honorary degree from ETH.

BL: I do. Yes. That was the first honorary degree I got. So ETH as you know is one of the top schools in Europe and a good science and technology school, so yeah.

And the other thing about the Turing Award is you get called on to do a lot of travel. So in the couple, two or three years afterwards, I traveled and traveled and traveled all over the world. And it still happens, though not as much as it did. So that's been a lot of fun. And my husband has come along with me, so we've been able to experience that together.

TVV: So where are notable places that you went?

BL: China. India. I was thinking… So we've been to the Far East several times. Of course we've been to Europe in many places. The only place I haven't been that I would really like to go is Africa. But yeah, it just seemed like there was a time there when there was so much travel. I think that is partly, when you get the Turing Award, you expect that this is going to happen and really you need to do that travel. It's sort of part of what's involved. Yeah.

TVV: Well, are there other things that we didn't cover that we should have?

BL: [laughs] It seems to me we've done a pretty good job of covering things.

TVV: Well, then let's… Thank you. Thank you so much for your time.

BL: And thank you for your time. It's been fun. Thanks.