PC Software Workshop: Technical - Development

Moderator:
Larry Schoenberg

Recorded: May 6, 2004
Needham, Massachusetts

CHM Reference number: X4621.2008

Table of Contents

PC Software: Technical - Development

Conducted by Software History Center – Oral History Project

Abstract: People involved in the development of early PC products talk about the methodologies and processes used to produce and test software. They identify the differences in the process used in the mainframe world versus what was used for early PC products. They also discuss the evolution of the development process as the PC industry matured.

Participants:

Name	Affiliation
Larry Schoenberg	AGS, moderator
Bob Frankston	Software Arts, Lotus, Slate, Microsoft
Evan Koblentz	Computer Collector E-Mail Newsletter
Mike Maples	IBM, Microsoft
Doug Ross	EECS Department, MIT
Scott Tucker	Lotus, KMT
Paul Ceruzzi	Smithsonian Institution, historian
Nathan Ensmenger	University of Pennsylvania, historian
Thomas Haigh	The Haigh Group, historian

Introduction

Schoenberg: This workshop is on technical development. I don't think it's precisely necessary to stick totally to this topic but I want to make sure we don't go into other people's areas. I'm Larry Schoenberg and we have as the historians Paul Ceruzzi and Nathan Ensmenger. Nathan is here. Paul is still taking a little break. The subject here is technical issues: development, testing and maintenance. And I think it's a very interesting subject. In the late 1950s I personally developed several testing tools to help people test programs and do retesting when modifications were made to programs.

One of the big problems we had, especially for systems software, was that the generation of the test data was invariably done by the same people who developed the software itself since no one else really knew what was going on. And of course, we all had great difficulty in perceiving

what kind of cases could come up and what testing was intended to do.

One of my best stories about testing is involves IBM. Despite the fact that everyone says that IBM projects were written by many people, it wasn't really totally true. Many pieces of software at IBM were written by one person, including the pilots in fact. And one of the areas that I wrote the software by myself was in this very area: debugging, testing, data generation, and so forth. Because we didn't have very good external storage, namely we had cards, it was necessary to write the program in as small an amount of memory as possible; whatever memory it used destroyed what was already there. So if your bug was exactly where this program went in memory, you never were going to find it.

I had this program which essentially loaded on itself; it just kept reloading itself. It only took 160 characters of memory, which today seems impossible. Though it took me a long time to write, it wasn't impossible at all. I was gone for about five years from IBM, and I awakened one night in the middle of the night and said to myself, "I know where I put a bug in that program." It turned out that in my program I replaced a character because I had this space limitation. But I didn't do it intentionally.

And since the program ran maybe a million times a year, it seemed puzzling that it was never recorded as a bug. Well, in examining the issue, it turned out that it wasn't reported because you could only replicate it by repeating the same program with the same data. But normally, if you had a bug and you found the bug, even though the program itself may have stopped, why would you go and repeat it with the same data? The end result was that it never got reported in millions and millions of uses, even though it must have happened plenty of times. The big clue, like in any program, is that you can't get the same results unless you repeat the program with the same data. But nobody ever tried to repeat it with the same data; or at least when they did, they gave up on it.

So I had really spent a lot of time on developing techniques for both testing and debugging; and even today I think it's fair to say there is very little quality debugging work that's going on. And it would be particularly interesting to hear someone who developed PC programs say that because we've said you guys developed some pretty buggy software over the years.

Okay. Let's start by doing what we did before and go around the room. Bob Frankston, would you like to start off with some statement about this subject?

Differences Between PCs and Mainframes

Bob Frankston: Well, let me start off by defending micro software.

Schoenberg: Defending buggy software?

Frankston: Yes. Why buggy software is so important to the industry

Schoenberg: It creates jobs.

Frankston: It does that – it creates jobs if necessary. The Six Sigma and all this debug software is a fatal mistake and it is why companies failed. Now, I'm not saying you want more buggy software. But I defend Microsoft Word, which is not the most bug free piece of software in the world. The reason is that, and this is true for the PC world in general, if you can survive a bug and keep it working and gain a lot more features, this is far better than having programs bug free to begin with. So if you don't worry about having some bugs and there's a low cost of ownership and the program does a lot, there ends up being a cost benefit tradeoff.

And the thing with the PC, unlike the mainframe, when you reboot it, your work is still there. I'm not saying it's perfect but it works. With VisiCalc, for example, I really had to do a lot of work to make it bug free because of my own financial constraints. But as we got more complicated it got harder. You can't eliminate all the bugs, so on your first try you should try to survive the bugs, and then you should try to eliminate them. So I think that this idea of debugging first is hopeless because it's too complicated, but it also means you're being too conservative and you can never move forward.

Schoenberg: Mike, you were on both sides of that world.

Mike Maples: My name is Mike Maples. I was with IBM for 24 years and I was with Microsoft for a number of years. At IBM I had a number of different jobs. One time I was product manager on the 3270 family of products. And at one time I was Director of Development for one of the product groups – a group that included ESD with all the small systems such as the printers and the industry systems. And then I went to Microsoft and I was responsible for the development of the application software for two years, and then in the last five years I was responsible for all of the products at Microsoft. So I spent some amount of time thinking about worrying about development processes and so forth.

I think there are two really significant differences about the PC industry as it grew up. One was it had a very open hardware, and hardware people were allowed to innovate in about any direction they wanted. And a reasonably high percentage of hardware or software bugs are a function of the hardware alternatives – just the number of permutations and combinations of disk drives and printers and video cards and video displays and sizes and so forth. I think it has been a real boon to the development of the industry to divorce the hardware and the software innovation cycle. And I think the thing that's allowed the hardware innovation to move so rapidly with the PC is the fact that you could throw something out there even though the software didn't support it and with some small amount of modifications or something you could make it acceptable.

I think the other really big difference was that, at least in the very beginning, the PC was one person, one machine, one application. And so the impact of a bug was relatively insignificant.

Schoenberg: Except for that person?

Maples: Except for that person. It didn't bring down the network, or it didn't bring down 1,000 users. And secondly, the users themselves had the ability to learn and adjust to the software. And that can't be done when you have 1,000 users trying to adjust and understand who else is running and what else they're doing and what they're doing simultaneously. But, as a PC, I know what I am doing. And if I know that it crashes, then I can just not do that if I choose to. I can choose not to use it or I can choose to avoid it by other means. As the systems get to be more and more acceptable or commonly used, I think that the world changes in terms of the expectation.

Bob made another point which I think is really true, and that's a question of function versus features versus performance and reliability. And there is the time in the life of things when people want it to do neat things. In the building of early cars, there were a whole of series of things, such as how fast they could go, that were more important than the fact that it always started when you got in it and that the heater always worked. Well, today the consumer demands that the heater works and the car starts. And so I think software will head there too.

Schoenberg: So, how did this affect development itself?

Mainframe vs. PC Development Methodologies

Maples: Well, I think if you look at the world in 1975, there were two development methodologies that evolved. There was the MVS development methodology that IBM had evolved to. You'd have three phases. You'd do the spec. Everybody in the world would sign off; 300 people sign off. Then you'd develop the code; 300 or 400 people would sign off. And then you'd test it, and then you released it. It was a two or three or four year development cycle. And that evolved over the debacles with MVS and MVT and MFT and the original 360 operating systems.

And I think those were the first really big software system architectures that were put in place. And then, you had the PC architecture in the early to mid-1970s, which was anybody could write anything and you did testing by letting the users try it, and then everybody submitted bugs back to the manufacturer, and then you debugged it. So you really didn't have a development process or test system.

Frankston: It's interesting because I grew up sort of on the side of high leverage developers because I was keeping track of data on the mainframe side of the Multics project. But what I was always trying to do was get leverage. In other words, what could you do over a weekend to get it to work. But it wasn't that you wrote any code – you had to spend years building tools. So with Multics, what I would do was debug interactive data but basically do it by myself. And I kept functioning; the goal was to keep going. On timeshare systems, the goal was it had to be up for use that day. If a feature worked, you could use it. The goal was to keep it always going.

And this is the approach I followed with VisiCalc from the very beginning. With VisiCalc you had something real to test; you weren't just testing a component in isolation. You were testing the real function. But more important, you were learning as you went along because you learned as you developed to make it what it was. You really didn't understand the product when you first shipped it. The old idea that you can have a spec and sign off was part of the strange idea of proving programs correct. I remember [Howard Olsen?] giving a talk at Lotus, and he explained the methodologies he used to debug a program. And so I asked the question, "What if you developed a program and the plane crashed?" Well, that's not just a software bug. And that's the flip side. There were games you would play to say, "Oh, it's not my fault," and there's a whole methodology that was fine except it wasn't reality. So you couldn't prove it was correct.

So the idea in the future is not that the system should run forever, but to build resilient systems. And I think this is sort of true of the PC in that it's not a monolithic thing where one slight bug will crash the whole thing. And if you look at the Internet, it's resilient too; if a site goes down, the rest of the net keeps working.

So the overall idea changes and this is a period of transition. In the old days, the legacy was the punched card and big program that does a job and produces a result. This is one model, and that's the model people think of in software. And really when you look at it, the nature of a bug change is not one that kills a system, but one where you can keep going; you try to limit the amount of damage done. You can't prevent surprises since you don't even know what a bug is. A bug is defined as the program doing something that the user thinks it's not supposed to do. It might be exactly what you had told it to do. But, it's surprising to the user. That's just as bad as a bug where it really just does wrong. So we have to really understand that it's not a single right or wrong in and of itself.

Thomas Haigh: I would be interested in getting some historical specifics.

Early Development Methodologies

Frankston: VisiCalc in particular used a development methodology that was exactly incremental. How the hell did I know what this program was going to do? We were figuring it as we went.

Haigh: You read what's been written about the early days and you have the impression of the people in the Homebrew Computer Club flipping switches on an Altair to make it play a tune that you could hear on the radio. There's paper tape. There's Microsoft BASIC. At least here are some basic tools. VisiCalc, I believe, was cross compiled from a larger computer.

So it would be interesting to hear some fairly nuts and bolts stuff about how software was developed and how debugging was practiced. From what you said, to some extent you start with a small maybe useful thing and keep going from there.

Frankston: Well, there are a couple of ways to look at that. There is one where you're just playing with things and I remember some of the Apple things were like that. So with a lot of programs there's no real direction. But VisiCalc had a direction. We had a path and we had a sense of the spreadsheet and we had to make some early design decisions. There were a lot of constraints. We had a small machine. And one of the major things was building tools. Let me just talk about this for a second without spending too much time on this.

It was like going to assembly programming because one of the things you learn after programming for years is that you've got to make the code understandable to yourself when you wake up in the morning because you're not going to know how you did it. And cleverness was severely punished. Readability was far more important than any cleverness. If you were going to be clever, you'd better be ready to explain what the trick is. This was something we learned in the 360 days. If you leave something in register 9 for more than one page, you know somebody is going to come in and reuse that register. They're going to look at the code and go through 20 lines and if it's not mentioned they'll figure it must be available so they could use it. So I was very concerned with readability for myself, let alone others.

Haigh: Yes. And this was written using your own custom sampling program?

Frankston: Yes. Basically, the idea of IF THEN macros in the assembly language was implemented. One of the things they did was a type of call return that compiled into a GO TO, but when you read the code, you knew you didn't have to worry where it went because control had always returned to the caller. So there were all these things to make it readable, but yet there were also other tricks. But I think the important thing was to make a very good guess at the beginning because once you paved that first path, it was very hard to change. So the basic

structure was good enough to go ahead.

Maples: I think you really want to talk about microcomputer development in the 1975 to 1985 time frame. It's a very simple subject.

Tucker: Not for us ... [LAUGHS].

Schoenberg: Go ahead. In what ways?

Maples: The problem is most of the innovation in software development probably really happened after 1985. I think there are two things that you have to think about. One is process. What process did people generally use to develop it, and what were the pieces that caused you to plan a project. And the second one was the tools that you used. The problem in the 1975 through the 1985 time frame was there were hardly any tools and there was hardly any process used. It wasn't until after that that processes were put in place by most of the development companies. Most of the microcomputer development before 1985 was done by very small organizations of two, three, four, five people, who used the workstation itself and some form of a compiler, and the testing was what they and their friends did at night. There wasn't really very much automated or thoughtful testing. There weren't real testers other than what was called gorilla testing, which is to take a function and see if you can break it. And I think that to really think about the development of micros, you probably have to go from 1983 or1984 on to look at tool sets that evolved and what were the various technologies that were in the tool sets, the editors and the development processes where there was some degree of precision in the development process.

Frankston: I don’t agree with you, Mike. We may have used this sort of gorilla style of checking in the beginning and we didn’t have the tools. But as we grew, we definitely had to become more formal.

Maples: Yes. I think that there were tools for development and there were QAs, but I think QA was a matter of taking a spec and trying to duplicate it on the machine. There weren't keystroke recorders. There wasn't good recording of all the tests you had done previously so you could run them again to see if it still didn't break. There wasn't even good logging of the data for product support or for passing back to the development process where it would be tested to make sure the same error didn't happen again.

Schoenberg: Doug, because you came from not only the mainframe world, but a world which was much more dependent upon accuracy, what did you about it?

Doug Ross: Well, with the first program I wrote, I couldn't get it to work so I wrote a program to find the bug in its mistake diagnosis routine. This was on Whirlwind and I had the

debugging cycle for that program. But the first time that that debugging program worked, I used it on the other program and discovered that the bug was one bit changed in the whole program. That's how brittle the software was. This was on Whirlwind with a 16 bit word and it was just one bit in one instruction that was between part A and part B, and it would never get past it because it was wrong.

And so I learned very quickly from that, and we went on and always used methodical ways to have modular structure even in the early days on Whirlwind. I wrote programs that were labeling the tick marks like on a ruler, using only three different kinds of instructions which were the ones that would reset as you did programming, jump in and out of the sub-routine.

When we got along further, we started to systemize those things and we developed a lot of technological methodology as we went along.

Schoenberg: You know, Bob, you made a comment that really hit me because I was one of those guys who was always writing clever programs and then discovered how disastrous they were. It's fine for you to learn for yourself, but if you have a group of people, how do you do that?

Frankston: The important point to structured programming is because of the interactions between disparate elements of a program. If you test one piece of code and you test another piece of code and they both work, the chance is pretty good that if you run them together the whole thing is going to work. But one of the problems is, and Word is a good example of this, you often don't have the luxury of keeping these totally isolated. We found some interesting bugs in VisiCalc where you find your [fence?] representation happens to coincide with the bit representation another program uses for another purpose. But to the extent you keep things decoupled and not interact with other elements, that's the best defense.

Schoenberg: How did you ever figure out what percentage of your code had ever been executed? I mean, because it's never been executed, you don't really know if it's bugged.

Maples: I don’t think anybody in the PC world before 1985 ever thought about that question.

Frankston: Yes. But because of the scale of programs in those days, the issue was more how to keep it to fit within finite boundaries, rather than a huge volume of code. So this is where you wound up with a few more clever things than you would have wanted.

But, there are other things. There are different configurations of machines and hardware. There's time. Therefore, one of the very strong things about the PC industry was the extent to which other programs in the PC world and the IBM world, the Microsoft world, would stay

working. Apple was not quite as concerned about this type of compatibility.

Ross: They also knew things about how you defined what you were trying to do with the small hit level. One of the things that was very instructive in my software engineering course (I taught the first one) was making an ordered list of values. And if you approach it just by thinking about what it is, you have to be concerned about the empty list that you begin with, and you have to worry about falling off the end of the list at the other end, and then you can figure out where in the list to go to get the value. But if you instead define your basic data structure for this ordered list to be a single node with an infinite value and a pointer to itself, then every operation is an insert. With every one you just swap the pointers. And so all of the separate roots and paths and so forth that you would think would be needed to handle all those special cases – they disappeared in the design.

Nathan Ensmenger: Doug, you mentioned software engineering which came up in our earlier session on accounting software, and it was interesting how little overlap there was between, developers and managers in the mainframe accounting firms and people who were trying to set up accounting software firms for the PC. And in mainframe computing discussions, you said that software testing methodology was well established by the mid-1970s. By that time they were talking about software engineering. They were developing these tools; they were paying a lot of attention. And then you have this discontinuity where the early computer PC firms are usually individuals. They're usually pretty small organizations. It's a different technology. It's a different culture. And I think in some ways, Mike, what you're talking about in 1985 is a kind of re-convergence of these, and it's not just technology. It's the markets that are changing, and it's bringing with it processes and culture. I'm most interested in different kinds of programmers and managers who have a different idea about how developing software should happen. And since you were at both IBM and Microsoft, which are very different companies in lots of different ways…

Tucker: Particularly in the middle 1980s.

Koblentz: That's an important point because in the mid-1980s, those PCs were as big as the mainframes were in 1970.

Ensmenger: But, I'm just wondering if you could talk about the other aspects of transforming this industry that aren't just the technical elements.

Reasons for More Strict Processes

Maples: Well, I think the first thing that happens is there are some failures. More often than not the failure is that people miss their schedules, probably more so than worry about the bugs. And so companies lose big ground in that. And the second thing that happens is the

tasks get large enough that you need more than one or two people on them. And so now you've got to figure out what are the communication mechanisms and the processes that are necessary so that two or three people can work together, or five or ten or twenty or some larger number. And then that leads to thinking about (generally it's not thought about until there is a failure, but then at least there is thinking about it) how can we make this more of an engineering project than an arts project or a science project.

I think PC software before 1985 was primarily individual art. The quality of the programmer who wrote the program (and that may be something that people are born with, not taught), was the difference between the great programs and the okay or not acceptable programs. And as the companies grew past the five programmers that they were able to get out of the basement of Berkeley or wherever, then you had to really start thinking about processes.

From a historical point of view, I think the most interesting thing about the subject is not that the PC guys didn't like the mainframe guys then because I think the PC processes today are a lot different than the mainframe processes that have evolved, and they were really geared to different problems. IBM had a three year cycle in their big operating system stuff and schedule was everything. They promised people things, and they had the hardware tied exactly to the software. They couldn't ship the new disk drives unless they shipped the software at the same time. And so it was a very tightly integrated system, and there was no room for failure on schedule. And so that meant that you built in a lot of buffers and a lot of contingencies and a lot of things so that everything would happen on the time. Now that might have been two years later than it should have happened, but having it all happening at the same time was better than having it two years earlier. But the PC guys had a different set of problems, and that was how to get out lots of functionality. There were an infinite number of functions and features that they wanted to do; and how could they get the most out in some reasonable, small amount of time.

Frankston: There is an important point to realize about the differences in our mainframe experiences. I grew up in the timeshare world where the product was services. What you were creating was a service. I was also building an OS which was a very different product and you needed to have SEs to support it. You didn't just ship the program. You shipped a person in the box. That's why the box was so big…you had put a person in it. And the users for the mainframes were very different.

The Use of Specs

Maples: If you go back and look at the development processes of 1970 to 1980, the high end was characterized by what IBM called the red book. And I'm sure there are copies of it around. It was about that thick, and it talked about the phase development process where you had a spec phase. Everybody signed off on the spec. You never changed the spec after a sign on. You weren't allowed to fix anything no matter what the market told you or what you learned. Then, you had a development phase for multiple phases – one, two and three – where you

developed the product, then you tested the product, and then you shipped the product. And the measure of success was that three years later you shipped exactly what you said you were going to do three years before.

Virtually every major software company did that although I wouldn't say that was universally true about application software. I think today if you look at most microcomputer software, they never have a finalized spec. And I think not having any spec is different than not having a finalized spec. And for the first products, I don't think there were any specs, period. And then, people would say, "I'm going to start writing it." If you saw a new feature at the trade show, you'd add it in and it would keep going like that.

Testing Methodologies

You kind of innovated in your mind more than you did in the spec. But I think the microcomputer guys as of 1990 were well on a path of defining a series of objectives for a release, defining what it was going to include and not include with some form of a spec, and then defining maybe three phases of work where you would go through the design, code, test and be ready to release a third or a fourth of a product at a time. And then, at the end of each time, you would reassess for your own schedule and then you'd decide which features do you throw out, or do you lengthen the schedule. And in some classes of software, primarily operating systems, linking the schedule is the right answer because you get all the people you support on it. And in applications, cutting out features was generally the way to go. You'd generally, for the customer expectation, for seasonality – Christmas, back to school – for whatever reason, you'd need to have the product ready to ship on some day. So you would decide a fourth of the way into the project that you had too many features to make it happen, and then you would just start dumping them way into the future releases, which is much better than the big red book process where at any one time everybody thinks they're 95 percent through. And all of the big tragedy failures and scheduling of software products is result from the fact that you're not testing two thirds of the activity. You're just testing some part of the code or some little micro part of the process, and all the things you don't know, you don't know.

Ross: With regression testing, though, you'd run it overnight then come back in and you know what the results had been for the previous day with respect to what had gotten through.

Scott Tucker: But, PC software was so interactive, and we didn't have a lot of these automated tools to do that kind of overnight testing.

Frankston: But, we had humans, which is sort of like the same thing. One of the points is that even by 1985, the large companies in those days – Microsoft, Lotus – did have a lot of ways of trying to do repeated testing on methodologies.

Maples: But, I think the difference was that you didn't freeze or change the spec. On the big projects, on an IBM project, you wanted to ship three years later exactly what you specked at the beginning. And on a microcomputer project, you wanted to ship what the market wanted a year from now. Or you wanted to ship the highest priority things that the market wanted, and that was going to change. That was going to be evolving because of what the competition did or because of what the hardware did. There is no way to think about what a three year spreadsheet would look like. You were worried about what the six month one was going to look like, and you change it three times in the six month period. And so the number of features and the number of capabilities were huge compared to the time you had to do it.

Frankston: But, on the other hand, after you shipped 1,000 copies into the stores, there was a lot of concern about discovering something after it shipped.

Maples: I think it was also after 1985 where the test organizations got the tools and that people had some sense about how bad it was going to be before it shipped. I think prior to 1985, most PC software was shipped without knowing how bad it would be, if it would be bad. They had written it; they thought they wrote it well. They tested it by however many people they had in the test organization, the QA, and they'd test as many features as they had. There would be a huge number of configurations they hadn't tested. There would be a huge number of combinations of features they hadn't tested, and they would always be surprised by the number of bugs they got or the number of problems they got.

Haigh: So at what point did the formalization of alpha and beta testing become established? And it also occurs to me that in some sense the end of alpha testing is equal to freezing the spec, isn't it? The idea is that in beta testing you don't introduce anything new

Maples: Well, alpha and beta testing was always popular in the PC industry. It was purely a marketing gimmick that they would try to get some number of enthusiasts interested in how good the program was going to be before time. And what you learn as a developer is you really aren't getting very good feedback during an alpha and a beta test. And so what Microsoft ended up with just after 1985 was first doing an alpha test. They would have a small number of people who were really good at giving feedback. And they'd call that an alpha test and the testers would get pieces of code that wouldn't work as a product. And then they had a beta where they'd ship out 100,000 copies; the main purpose was to get all of their most important customers to be in the know about the new product before it shipped. In most cases, if you had a process, you would be debugging your product and you'd get these random reports of bugs from the beta testers that you either had already solved, or you hadn't researched that piece, or their version of the code was already six weeks behind time, or you wouldn't get any feedback at all.

I think beta testing as a testing methodology is probably very overrated. And I think the only decent betas now involve the operating system because there are so many variable

configurations or the ones sent to developers. But beta testing of applications is primarily used as a marketing tool.

Tucker: Well, I wouldn't disagree too much with that, but there is some issue about if you get it out to a lot of people in beta, that does give you some security that if there are different configurations you're going to find out before the product goes to market. So at least there is some safety value in doing that.

Frankston: Yes and then there are those of us who even distributed the product before beta testing anyway.

Maples: But, the problem with doing that is that you have to assume that your users are going to try a piece of code that you're not going to guarantee it works, and put some real work on it, and be wiling to let their real work go south.

Frankston: Yes, that was the risk.

Maples: And I don't know how many companies' beta stuff you get, but I've had lots and lots of beta software, and I'd say 80 percent of them I never put on the machine.

Frankston: But, on the other hand, I've had the opposite situation where I'll often run the beta because as bad as it is, it's better than shipping the product.

Ross: Yes. So you don't want to rely on that as a methodology for determining things, but it does give you occasionally some good feedback and is, therefore, worthwhile.

Haigh: And when you say it's always been part of the PC industry, does that mean it was common by 1979 or 1980?

Frankston: Yes. Remember that a lot of the PC people were in programming and were familiar with the alphas and the beta tests and gained experience with the early test cycles. So the concepts were not new.

Tucker: I joined the PC business in 1983, and it was common; it was established by then.

Maples: I think it was very common. I think the depth of the distinction between a shipping product and a beta was a matter of the calendar more than a matter of the quality of the code.

Frankston: Yes.

The Significance of Quality Programmers

Ross: Can I ask a quick question to you, Larry? You mentioned earlier sort of discrediting cleverness, or discouraging cleverness. Did that ever come back to haunt the industry by having so much rigidity? What I mean is by encouraging more structure versus tricks, didn't you have all those extra lines of code you needed to go through to find bugs?

Schoenberg: It wasn't my point, anyway, but I did comment on it. I mean, it is true that the cleverness was more a perception of efficiency than it was of quality of code per say.

Maples: I think the thing that nobody understands even today in the world is that a really good programmer was 100 times better than just a programmer. And so no matter what process they used, they are just a lot better. There are some people who are just a lot better. And you know, it's kind of like maybe tennis stars or something. And so process is really geared to having a lot of people that are in the B and C category and making sure that their pieces work, more than getting the A plus guys to do this hundred times x. You know, those guys will go off and do something else. They're working on something else. They're not going to be working on Microsoft Word.

Frankston: The problem is the process. You would want to run away screaming from the groups that had too good a process. And yes, Word was a process group, and a lot of processes don't work.

Haigh: I've got another question, then. It's to do with process, but also the emergence of this idea that we have today: there's Word 11 and then 18 months later there will be version 12, and then there will be 13. And no one here would bet against there being a Word 25 or 30 at some point in the distant future. So we know that the next version will have a few extra features and there will be some bug fixes released over the Internet in between. Now, one of the things that really struck me this morning was hearing that after Lotus 1-2-3 Version 1 became the most successful product in the history of the industry, they said, "All right. Well, we did 1-2-3. Now we're going to do Symphony," and there was no 1-2-3 Version 2.

Maples: It was really 2.1 that was the big winner.

Haigh: So at what point and how did this idea that there is going to be another version of the product with some more features come into play?

Tucker: Well, I think you've touched on it to some degree here with a couple of comments. One is that initially in the PC business it was an individual or two who were

extraordinarily talented who did the entire product. And the products that survived and did well in the marketplace were those that were done by these outstanding engineers. They were outstanding but not even necessarily engineers. They might have been self taught, but outstanding.

Maples: They overcame huge barriers in small hardware, limited capabilities, and made it acceptably fast.

Tucker: Now, at Lotus, there was a big problem making the leap from that to a team, and that wasn't done terribly well. And so right away there was an issue there, but this concept of continuity of product...

Ross: We had continuity of people.

Tucker: Well, Lotus had neither.

Follow-on Releases and Products

Frankston: No, but with VisiCalc, we always knew there would be later versions. But I think that discontinuities were a factor. I think there was always the attitude in the companies that they would keep the product going to the end.

Tucker: Well, let's look at 1-2-3 Version 1 to 2. The product that shipped as Symphony started out as 1-2-3 Version 2 in internal development. I remember looking at an early development version of 1-2-3 Release 2, which is what it was called, and it looked a lot more like Symphony.

So I would say this, and this touches on something you said before, Bob, which is that you're developing the product and you don't really know what it's going to come out as. It's like writing a novel and not quite knowing where the plot is going to go.

Frankston: And I think the real mistake was making Symphony seem so different than 1-2-3.

Tucker: But the point I'm getting at is it started out intending to be the next version of 1-2-3. And by the time it got close to where they were starting to talk about it publicly, they came to realize that it had grown into something else unintentionally.

Haigh: But, it was also clear to the mainframe industry by that point, that backward compatibility and the value of an installed base were understood, and by this point there were already products like IMS and CICS that had been around for a long time.

Whereas in this situation a company can have the most valuable product in the entire industry, and they just say, "Well, that was nice. Now we're going to replace it with something else." Well, that's why being compatible, doesn't have the same importance in the PC industry.

Tucker: That's why they renamed the product to Symphony, though.

Maples: I think there's an easy answer to that. The answer, I think, is that industries mature to understand the customers' requirements. In the beginning, they look at it as science projects or fun. The original products were done by guys who were having fun trying to do what they liked to do. They threw them out there. Some of them got successful. They didn't know why. They didn't go study the customers. They weren't trying to respond to the customers' needs. They were trying to do something else that was fun.

And then, ultimately, the reason there's always going to be another release is you you've got the revenue profit aspect. But, just as important, the more people use it, the more they're going to tell you about things you could do differently, whether it's improved to eliminate customer calls on problems, to add function that the customer wants or to make things easier to use. You know, I can't see that you could ever have the perfect one.

Ensmenger: It wasn't just Symphony. There was another program called Framework from Ashton Tate that tried to do multiple things. I'm just wondering, did Microsoft have one that they cancelled?

Ross: Context MBA

Maples: Microsoft did two things. We did a product called Works which was all things to all people; it was integrated to keep it simple. But the reason we wanted to do that was so we wouldn't have to sell the expensive products cheaply to the OEMs.

We recognized that we had a value and that we had a portfolio strategy. We had three operating systems, five applications. There was a portfolio and we were trying to fill in all the checks. And then one day we had the checks filled in and we said there's value in the portfolio greater than the sum of the parts. And while we always care about having the very best product per category, there is some weight in the bundle or in the sum. And so we came up with the concept called Office.

Koblentz: Which is fine, but I think that even that glosses over the point, which is Microsoft's integration strategy. It's called Windows.

Tucker: Nobody else had that.

Frankston: The 1-2-3 and Symphony thing I think was atypical.

Koblentz: But, Framework was another version of that.

Frankston: No, Framework was started from scratch.

Koblentz: It's the notion that integration is important.

Frankston: I agree with that. I just don't want to dwell on that. Every company did want to keep this franchise, to keep evolving the products. There were the two integration studies. Remember, the PCs are small, having separate components. That's why you had a program that was trying to do a lot. Now, before Windows, there was also Gem. There was the character integrated Window thing...

Ross: USD PASCAL...

Frankston: Right. But, the point of what I'm saying is that everyone wanted commonality. I remember getting bad reviews for Express because I didn't use the slash as the command key to word processing; that's what people expected. And what happened was Windows became the standard and everyone wanted that type of interface and commonality. So, as we rode Windows out, you weren't just running an operating system environment. You also gave us the GUI methodology. In the Windows world you became beholden to the metaphors. Now Office went a step further and had even deeper metaphors across it. And Office still has components maintaining an identity, which is very different than the earlier integrated products like Works, Symphony, Framework, etc.

Maples: Well, I think that Lotus fooled themselves a little bit because their genesis of 1-2-3 was an integration of graphics, a little bit of database, and a spreadsheet. So if that's good, then why isn't an integration of a word processor and some other things even better?

Tucker: Exactly.

Maples: And I think that it was just a strategy error.

Frankston: Well if Symphony were the proper superset of 1-2-3 so that the transition worked smoothly, it might have succeeded.

Maples: But it wasn't a proper superset.

Frankston: No, it wasn't; that's what I'm saying. But, if it were, if they did a clever job so

you didn't have to learn the other features to grow up, it might have actually been a success. In all of this, you have some of the accidental properties from the basic issues.

Haigh: In terms of recognizing the value of the installed base and the idea that a product to some extent has a very long life, isn't it true in the case of VisiCalc that the opportunity to market domination only existed for 1-2-3 because VisiCalc hadn't been extended.

Frankston: No, no. This is the whole discussion. Since Dan's boss is here, I can put my bias away because VisiCorp decided that it was all about marketing and these guys went and decided to commit what became suicide, then came 1-2-3. As a matter of fact, if they hadn't ticked off the VisiCalc product manager, you wouldn't have had 1-2-3. It was a whole set of preconditions that made it happen.

Maples: VisiCalc was a funny company because it wasn't a company; it was two companies. It was a development company and a marketing company, and they didn't necessarily have the same goals.

And the second thing that happened is the PC platform changed from an Apple to an IBM in the business world. And so you had a much bigger memory set and a bigger disk, and 1-2-3 took advantage of that.

Frankston: Well, more to the opposite point of view, 1-2-3 failed at their ambitious goal on the Apple. They said, "What the hell. We've got nothing to lose by requiring more machine." And the surprise was we never thought people would pay money for this. So the fact that 1-2-3 required a huge 128K machine was surprising; they got away with it. You know, we keep underestimating the value of this.

Tucker: Because people didn't understand Moore's Law.

Frankston: But, that doesn't have to do with Moore's Law. They found it valuable.

Maples: It didn't take long for machines to catch up so that everybody had Version 1.

The Microsoft Approach

Haigh: I'd be interested to hear the Microsoft angle. And Microsoft, at least through the early 1990s, had the reputation as a company that didn't have so much brilliance floating around, didn't necessarily make the best products, but did always stick with it to do the three versions it took to get a good product. At least the outside impression is of discipline, of sticking through the things and knowing that it will take various versions to get things right. Do you think that was actually true of how people inside the company thought at the time?

Maples: I think it was. I think that the other strength and weakness of Microsoft is that they would pick a strategic direction and keep going for a long time. In the early 1980s, you probably don't even remember the products, but there was a product called Multi-Plan. There was a whole series of products.

The Microsoft belief was that there would be a lot of PCs that would be somewhat different from each other; and that there would be a Wang, there would be a DEC, there would be a TI and there would be an IBM PC maybe incompatible in numbers of areas. And so you wanted to build a core of your application, they were called multi-tools, that could be easily moved across these different platforms. The surprise was the IBM PC architecture wiped the others out. So here Microsoft had its strongest technical point, its ability to move across platforms, when it wasn't necessary. So Microsoft did very poorly through the mid-1980s with their applications. They had a broad set of applications, and none of them were even second or third; most of them were in third and fourth place in the market.

The one thing that Microsoft did in the early 1980s that proved successful was that they were aggressive at building business apps on the Macintosh. They saw the Macintosh as a powerful machine, not as a toy machine. In the original machine, you remember, the big programming feat was to get it to say hello on the screen in some scripty font.

And so you looked at what Lotus did and what the other business software companies did on the Mac. They followed Jobs under this little bit of a frivolous application, and they also designed for the very small first version of the Mac. And Microsoft's versions came out a year or so later, but they had a lot more power. They had the kind of a PC level of power with the Mac user interface. So Microsoft learned about graphical user interface before 1985, where virtually no other company applied graphical knowledge to the business world. You know, Jazz and most of the things you could buy for the Mac were not really business quality functionally rich kinds of products.

So Microsoft then came out with Excel and started working with Word for Windows and the other applications around Windows, but they couldn't sell them because Windows was so poor. The biggest problem with Excel was the fact that it ran on Windows. You had a 640K machine and you didn't have enough memory. And so Microsoft ended up building a number of unique

compilers and other tools to handle building graphical applications in a small system.

And so Microsoft was sitting there in the 1985, 1986 timeframe with a lot of knowledge in graphics, and they believed that graphics was going to be significant. So where they first had the belief system and the multiple incompatible systems, they changed and the belief system became based on the graphical user interface. So everything was geared to that probably three years ahead of time. And this was a case where Microsoft had nothing to lose. We were kind of like where Lotus was. We didn't have anything to lose. And we had everything to gain, too.

And Lotus and WordPerfect and the various other competitors (a) didn't understand the market as well and (b) had every reason to like it the way it was. You know, they were getting $500 a copy for a non-graphical kind of thing so why change.

Tucker: And further, they would see part of their design in Microsoft's products, as Bob mentioned earlier, which made them bristle.

Maples: I think that being focused on some strategy was important for Microsoft, and then there was this whole idea of ... let me call it a customer feedback loop. And you know, it's kind of a joke to say, "Well, they don't get it right until the third version." I would suggest in most cases engineers don't understand real people until about the third time they've heard them say it. So I think it makes more of a law than a phenomenon.

Frankston: Well, there's also reinforcement because if you had a great DOS product, all the users would say make it a better DOS product.

Maples: But, I think the thing is you release a product and you listen to the customers. You learn that they want some new feature. You see the competition. You see what's happening in the world. And so it takes you a while to figure out, starting from an engineer viewpoint of what's a great spreadsheet, what is an accountant's view of what a great spreadsheet is. And I think that's why Microsoft is so good at that. I mean, that's the reason they've been so persistent at products. And Microsoft hasn't ever killed a product. They keep products way too long.

Single Platform vs. Multi-Platform Products

Tucker: Can we stay on this topic of multi-platform versus single platform for a minute? That featured prominently in Lotus's demise as well, I would say, because, as Mike points out, Microsoft implemented Multi-Plan in such a way that it was multi-platform. It could be ported easily, and so on. But then they implemented Excel really using Lotus's example of focusing on one hardware platform – be blindingly fast and be great on that one platform. Microsoft took a lesson from Lotus in doing that. Lotus then, in trying to implement 1-2-3 Release 3, went back

to what Microsoft had abandoned as a strategy, which was to try to make it multi-platform. The reason being that Lotus wasn't sure what direction the industry was headed.

Frankston: Now, Release 3 was purely a DOS product.

Koblentz: No, no, no. Release 3 ran on the IBM OS/2.

Tucker: But, you remember that from the very beginning of 1-2-3 Release 3, one of the main reasons it was written in C was it was explicitly being designed to be a code base that could be easily ported to a variety of platforms.

Frankston: Yes, but it wouldn't have been so bad if the graphics front end were given enough priority.

Tucker: Well, maybe that's true, Bob, but I prefer to believe that we probably could have shaved a year off the development cycle and probably could have shipped an appropriate product in an appropriate time if we had decided not to position this product to be ported to a lot of different platforms.

Maples: Every design decision has two effects. One effect is how important is it, and the other is what you didn't do because you did it.

Schoenberg: You know, all of this discussion has somewhat of a character of after the fact rationalization. I remember when Jim Manzi came to the ADAPSO conference about 1-2-3 and I had to introduce him. He was sitting next to me during lunch, and he said, "Larry, you don't realize how difficult it is to develop a multi-platform machine." And I looked at him like, "What, are you crazy? I mean, what the hell do you think all of us mainframe guys have been doing for our whole lifetime?" I was so stunned by the idea. Not that he had the problem, but that he had no idea that this was a common problem, boggled my mind. And so my view is that none of these products really ever recovered from not understanding what it was when they went from a small program to a big program. It's not just the integration. I mean, I look at some of these micro products of this time and they're bigger than any program I ever worked on. And I worked on big programs.

So suddenly they went from this one user, very dedicated application, to one across multi-platforms. They had no idea what the problems were. If philosophically one can understand the idea that the product you develop is going to have bugs, it's not so terrible when it crashes. That's okay once you have a market position, but you can't do that without a market position. So when you introduced a new product or you went to an area where people were not as comfortable, it was devastating when these things crashed. And I realize you're being somewhat flip in your comment because I know you weren't that flip at the time, but...

Frankston: Right. Software Arts was compulsive about being bug free and that cost us. When you speak about Sidekick versus Spotlight, Sidekick would destroy machines and do terrible things with it, where Spotlight was amazingly strong and robust. Well, we made some interesting discoveries that the dealers really didn't want to make money. They didn't know enough about this. And there was this guerilla attitude of running Sidekick. You were an explorer and an adventurer.

Maples: I think that the microcomputer software industry hardly hit business by 1985. So if we really want to talk about 1975 to1985, we're really talking about selling products in low volume to the hobbyist. And that's kind of what the market was through 1982-1983. And then in 1984 and 1985 it started becoming a business market, and the software development started evolving to that, but most of the real products and evolution happened on the very tail end of this.

Schoenberg: We're really going to have to quit very soon, and obviously the conversation has gotten into a very few things. So if we could just take a minute for anyone who has a question.

The Overall Trend

Ensmenger: If we weigh everything we've learned over the last 20 years in bug extermination versus the increasing complexity of today's products, are we anywhere near closer today to bug free software?

Maples: I think that there have been technologies evolve that have taken bigger pieces of code and maybe smaller pieces of code and have them work together. And there are probably less bugs per 1,000 lines of code or some of those measures. But, the number of lines of code in total has grown fast.

Ensmenger: Yes, exponentially.

Maples: Now, the question is will the average user of microcomputer software today find fewer or more bugs than they would have ten years ago? They find a lot fewer bugs today. Their system won't crash as much. The thing won't be nearly as consequential. And so, from that measure, while not perfect, I think it's infinitely better than it was.

Is there a way to get rid of all the bugs? I think there are probably as many bugs as there ever were, and it's controlled by controlling configurations, controlling a number of things.

Frankston: But, I will say several things have affected this. One is the switch to more robust operating systems. Windows was cobbled together. And because XP is now a real

system, we have some degree of isolation in certain things. The other thing involves the switch in programming languages. The C++ and all these types of programming languages are basically like overgrown assembly languages. It's amazing the stuff worked at all, and a lot of the viruses you see are consequences of those languages. But the tools have improved vastly and have really kept us in line despite the complexity. And I'm seeing limits on this, but the systems are much better. The problem is there are so many devices and things that it just exposes some of the problems with the interactions. It's amazing it works so well.

Schoenberg: I'm going to have to try and end this. We obviously created an opportunity for another session, but we're not going to get it right now. And we're supposed to end and give you a chance to, if not recover, at least reassemble into other places ... [LAUGHS].

So I want to thank you. I think obviously we've had people who have been very germane to this issue. There were certainly other possible subjects that I had a vision we could talk about, but I think this was as important a one as we could possibly have done. There's no hope of covering them all.

I want to thank you all.

