The Father of C# on the Past, Present and Future of Programming

Anders Hejlsberg is not resting on his laurels. He's off solving a new problem: Finding a way to query XML and other data using .Net-based programming languages.

Anders Hejlsberg is one of Microsoft's handful of distinguished engineers. He is known for having developed the Borland Turbo Pascal compiler and for having been chief architect of Borland's Delphi technology. Hejlsberg left Borland, where he last served as chief engineer, to join Microsoft in 1996. Since joining Microsoft, Hejlsberg's greatest claim to fame has been fathering the C# programming language. Originally code-named "Cool," C# was designed to be Microsoft's Java killer.

Hejlsberg chatted in June with Microsoft Watch Editor Mary Jo Foley and eWEEK.com Senior Editor Darryl K. Taft at the recent Microsoft Tech Ed conference in Orlando about the past, present and future of the C# programming language – among other programming-language-related topics.

In (Microsoft VP) Paul Flessner's keynote at Tech Ed, we heard about how Visual Studio and SQL Server and BizTalk Server are getting more and more tightly integrated. What are the implications of that increasingly tight integration for tools and languages?

For tools, it's that the cockpit you sit in gets more and more capabilities. My particular interest for the past couple of years has been to really think deeply about the big impedance mismatch we have between programming languages, C# in particular, and the database world, like SQL — or, for that matter, the XML world, like XQuery and those languages that exist there.

If we just take languages like C# and SQL, whenever I talk to the C# programming crowd — and I tried it in my session (at Tech Ed) — I ask them, "How many of you access a database in your applications"? They laugh and look at me funny, and then they all raise their hands.

So from that I take away that when you're learning to program in C#, you're actually not just learning to program in C#. You're also learning SQL. And you're also learning all the APIs (application programming interfaces) that go along with bridging that gap. And you're learning a whole style of writing a distributed application. But interestingly, we've come to accept that that's just how it is. But it doesn't necessarily have to be that way. The two worlds are actually surprisingly un-integrated.

On the tools side, we're making tremendous progress on getting deeper integration between these two worlds. But I think on the language side, we also could make an enormous amount of progress.

So, there's so much stuff that we could do there to make these worlds better integrated. And that's what we've been thinking about deeply over the past couple of years. And I think you see some of the fruits of that thinking in the "Whidbey" release. Specifically, a feature like generics (which will be part of Whidbey, or Visual Studio 2005) is not only a great tool for programmers because it allows you to parameterize your types and have much more code sharing, and get better compile-time type safety and all of these things that are goodness. But it also strengthens the type system, or makes it more expressive.

In terms of other languages, I've noticed that (Mr. Indigo) Don Box and (Mr. Avalon) Chris Anderson have been writing a lot lately about Ruby and Python. Where do you see Microsoft playing in that space?

We're doing a lot of work to make .Net an even better platform for dynamic languages.

I think interestingly that there's sort of a resurgence of interest in dynamic languages. When I look at it, I sort of see two different things there. And sometimes I think people confuse some of the advantages of dynamic languages. For example, some people say I love writing in Python, my favorite dynamic language, because my code is much terser and it's much easier to write, and so it goes much quicker. And sometimes people say, I think it's OK to pay the price of no type checking to get that terser code. Then I say, is it really necessary to have such strong typing in the name of tersity? I don't think it is, necessarily.

I actually think that an even better world is a world where (a language) is terse, but it is still strongly typed. And I think it is quite possible to do that. Some of it, in its infancy, is in some of the type inferencing capabilities we have in C# 2.0, where, effectively, we infer the type of the variable from the way you use it. And there's much more we can do there.

Would you go so far as to create another new language to be able to obtain this?

I don't think there's any need to create another language at this point. Once you create another language, In the name of solving one problem, you buy yourself nine other problems. I think we can very naturally extend the languages that we have so that you can do it.

In regards to Visual Studio, do you see all the languages that are supported now evolving in parity? Will you continue to do to Visual Basic everything you do to C#?

I think it is practically impossible for us to evolve in lockstep. Innovations will happen in one space and they will turn out to be great for users over here, and we'll adopt in another. But we will certainly always share all the information we can and attempt to have as much parity as we can. But I don't think it is either meaningful or possible to guarantee complete parity, just in the name of parity.

I do think there are some differences in the profiles of VB (Visual Basic) programmers and the kinds of characteristics they're interested in, and C# programmers, who are probably a little bit more interested in the bleeding edge of language technologies.

I was told — and I could be wrong — that the next version of C# (3.0) borrows a lot from FoxPro....

No. I wouldn't say that it specifically borrows from FoxPro. As I have said before, the area where we're already laying some groundwork for in Whidbey (Visual Studio 2005) is this big and largely unexplored area of deeper language and data integration. And, of course, FoxPro is a language that has been there, but also, like dBase and all of those languages, called 4GL languages — though I don't really know what that "4" meant — I think they show how closeness to data is tremendously useful for a certain class of applications.

But unfortunately, I also think that those languages, for whatever reasons, never got accepted as mainstream programming languages, largely. They lacked some capabilities that programmers, generally speaking, want.

What I'm trying to do is I'm trying to do the data thing without throwing the baby out with the bath water. And I'm being very conscious about that. Whatever we do to C# has to feel completely natural to C# programmers.

FoxPro comes at it from a different angle. They have their own run-time infrastructure and they are not hosted on .Net the same way as the other languages are. So that's the ball they have to chase. C# is already on what we think is the right run-time infrastructure but is lacking the capabilities of deeper data integration. So that's where we're looking.

The teams are actually right next to each other. I know all the folks on the FoxPro team. So we talk a lot. And they have a lot of experience in this space. I think our approach in C#, though, is going to be a little bit different. FoxPro is very tightly married to your data being data in a database. When I talk about data in this conversation, I mean data at large. It doesn't necessarily have to come from a database. It may as well come from an XML document or just be something that you moved off a bunch of objects and now you want to do set and query operations on these graphs of objects. So in that sense, we're sort of coming at it in different ways.

I'm curious about how you work. Do you work the same way as if you were an author writing a novel? Do you outline the 'novel' first, so you know where you want to go? Or do you just write 'chapter' by chapter?

I think you pick a big goal. Let's just say for the sake of argument that I want to make it as easy to program data in C# as it is in FoxPro. Or pick whatever goal you'd like. I'm not saying that is a particular goal. And then you start learning. You do your first approximation of the problem. And then you learn a bunch from it.

You start out by building what you think is the solution. And then from the solution you learn how there's really a more general solution to the problem…that the solution you built is just a specific instance of the more general solution. Then, effectively, you push yourself a meta-level. And that's typically how language extensions are like that. I'd be completely adverse, for example, to a language extension that tied us —(one) that said, we're going to do data integration and it's going to be SQL with SQL Servers and that's what we're going to jam into the middle. Well, that would be the kiss of death for C#. Because then, it's no longer a general-purpose programming language.

So it's that level of generality that you have to look for in order to insure your instance of a solution that you build alongside isn't right, then you can just throw it away and build another one and you're still good. Otherwise, what you end up doing is aging the language. And that, effectively, is what kills programming languages. When they've accrued so much cruft, they seem old and tired.

I've been interested in some of the various Microsoft Research projects around the various "Sharps," like X#, F#, Spec#. Are things that Microsoft is learning in those projects going to change what you're doing? How?

Oh, absolutely. Let's take generics. I think that's a wonderful example. Generics started out as a research project done by Don Syme and Andrew Kennedy at Microsoft Research in Cambridge (UK). These two researchers effectively took the C# language and the .Net platform and added parameterization of types on top of it. They modified it. They wrote two papers about it and gained some experience with it.

Their work was very credible and very applicable, since they were working with the same bits. So, all of that work worked gradually into the production (Visual Studio 2005) code. They ended up helping out and working on the production implementations of this stuff. I think the prototype (of their work) was called Gyro when it was built. This was built on the Rotor code base. It was, to me, a wonderful example of technology transfer from research into a product. I think Don and Andrew are great researchers and very pragmatic engineers. That's a great combination. I wish there were more like that.

Don is now working on this thing called F#, which is taking a deeper look at functional programming languages on the .Net platform. And certainly a lot of the stuff we're doing with C#, going forward, will learn from — and is already learning from — functional programming languages.

It's sort of interesting. In many ways, I see my job as making sure you keep up on what's happening in research and academia and bridging to real-world technologies. Because when you look at functional programming languages, like Haskell or ML or whatever, it's like, 'Oh my'! To a lot of researchers, syntax is just a necessary incarnation of semantics. Really, what they're interested in is semantics. And who cares what notation. You can make another one tomorrow, right? Well, unfortunately among programmers, syntax matters. You can see that in the religious wars over whether it should be VB (Visual Basic) or C# — which, effectively, comes down to what syntax do you prefer?

So in that sense, I'm an engineer at heart. But I'm keenly interested in research. So, yeah, F# and Spec# is stuff that we look at and evaluate all the time. We meet with the researchers. Our C# design meeting, which is a series of meetings that have now been going on for six years in the same room, in the same time slots, we have researchers come regularly and give presentations on what they're doing and vice versa. Those are all technologies we look at. C-Omega (formerly known as 'Xen' and 'X#') is another one. Some of the members of that research project are actually now members of the C# design team.

What are the general programming language concepts and trends that interest you right now, in the grander scheme?

Generally speaking, it's interesting to think about more declarative styles of programming vs. imperative styles. A lot of the stuff I've talked about are actually specific instances of that. Functional programming languages and queries are actually a more declarative style of programming. You sort of declare what you want to have done, but not exactly how you want to have it done. And in many ways, we've educated generations of programmers to think not just about what, but about how, and to explicitly state how in their programs.

In many ways, programmers have to gradually unlearn that and learn to trust that when they're just stating the "what," the machine is smart enough to do the "how" the way they want it done, or the most efficient way.

Source: http://www.microsoft-watch.com/article2/0,2180,1837433,00.asp