How Microsoft Lost the API War

By Joel Spolsky

Although there is some truth to the fact that Linux is a huge threat to Microsoft, predictions of the Redmond company's demise are, to say the least, premature. Microsoft has an incredible amount of cash money in the bank and is still incredibly profitable. It has a long way to fall. It could do everything wrong for a decade before it started to be in remote danger, and you never know . . . they could reinvent themselves as a shaved-ice company at the last minute. So don't be so quick to write them off. In the early 90s everyone thought IBM was completely over: mainframes were history! Back then, Robert X. Cringely predicted that the era of the mainframe would end on January 1, 2000 when all the applications written in COBOL would seize up, and rather than fix those applications, for which, allegedly, the source code had long since been lost, everybody would rewrite those applications for client-server platforms.

Well, guess what. Mainframes are still with us, nothing happened on January 1, 2000, and IBM reinvented itself as a big ol' technology consulting company that also happens to make cheap plastic telephones. So extrapolating from a few data points to the theory that Microsoft is finished is really quite a severe exaggeration.

However, there is a less understood phenomenon which is going largely unnoticed: Microsoft's crown strategic jewel, the Windows API, is lost. The cornerstone of Microsoft's monopoly power and incredibly profitable Windows and Office franchises, which account for virtually all of Microsoft's income and covers up a huge array of unprofitable or marginally profitable product lines, the Windows API is no longer of much interest to developers. The goose that lays the golden eggs is not quite dead, but it does have a terminal disease, one that nobody noticed yet.

Now that I've said that, allow me to apologize for the grandiloquence and pomposity of that preceding paragraph. I think I'm starting to sound like those editorial writers in the trade rags who go on and on about Microsoft's strategic asset, the Windows API. It's going to take me a few pages, here, to explain what I'm really talking about and justify my arguments. Please don't jump to any conclusions until I explain what I'm talking about. This will be a long article. I need to explain what the Windows API is; I need to demonstrate why it's the most important strategic asset to Microsoft; I need to explain how it was lost and what the implications of that are in the long term. And because I'm talking about big trends, I need to exaggerate and generalize.

Developers, Developers, Developers, Developers

Remember the definition of an operating system? It's the thing that manages a computer's resources so that application programs can run. People don't really care much about operating systems; they care about those application programs that the operating system makes possible. Word Processors. Instant Messaging. Email. Accounts Payable. Web sites with pictures of Paris Hilton. By itself, an operating system is not that useful. People buy operating systems because of the useful applications that run on it. And therefore the most useful operating system is the one that has the most useful applications.

The logical conclusion of this is that if you're trying to sell operating systems, the most important thing to do is make software developers want to develop software for your operating system. That's why Steve Ballmer was jumping around the stage shouting "Developers, developers, developers, developers." It's so important for Microsoft that the only reason they don't outright *give away* development tools for Windows is because they don't want to inadvertently cut off the oxygen to competitive development tools vendors (well, those that are left) because having a variety of development tools available for their platform makes it that much more attractive to developers. But they really *want* to give away the development tools. Through their Empower ISV program you can get five complete sets of MSDN Universal (otherwise known as "*basically every Microsoft product except Flight Simulator*") for about $375. Command line compilers for the .NET languages are included with the free .NET runtime … also free. The C++ compiler is now free. Anything to encourage developers to build for the .NET platform, and holding just short of wiping out companies like Borland.

Why Apple and Sun Can't Sell Computers

Well, of course, that's a little bit silly: of course Apple and Sun can sell computers, but not to the two most lucrative markets for computers, namely, the corporate desktop and the home computer. Apple is still down there in the very low single digits of market share and the only people with Suns on their desktops are at Sun. (Please understand that I'm talking about large trends here, and therefore when I say things like "nobody" I really mean "fewer than 10,000,000 people," and so on and so forth.)

Why? Because Apple and Sun computers don't run Windows programs, or, if they do, it's in some kind of expensive emulation mode that doesn't work so great. Remember, people buy computers for the applications that they run, and there's so much more great desktop software available for Windows than Mac that it's very hard to be a Mac user.

And that's why the Windows API is such an important asset to Microsoft.

(I know, I know, at this point the 2.3% of the world that uses Macintoshes are warming up their email programs to send me a scathing letter about how much they love their Macs. Once again, I'm speaking in large trends and generalizing, so don't waste your time. I know you love your Mac. I know it runs everything *you* need. I love you, you're a Pepper, but you're only 2.3% of the world, so this article isn't about you.)

Sidebar
What is this "API" thing?
If you're writing a program, say, a word processor, and you want to display a menu, or write a file, you have to ask the operating system to do it for you, using a very specific set of function calls which are different on every operating system. These function calls are called the API: it's the interface that an operating system, like Windows, provides to application developers, like the programmers building word processors and spreadsheets and whatnot. It's a set of thousands and thousands of detailed and fussy functions and subroutines that programmers can use, which cause the operating system to do interesting things like display a menu, read and write files, and more esoteric things like find out how to spell out a given date in Serbian, or extremely complex things like display a web page in a window. If your program uses the API calls for Windows, it's not going to work on Linux, which has different API calls. Sometimes they do approximately the same thing. That's one important reason Windows software doesn't run on Linux. If you wanted to get a Windows program to run under Linux, you'd have to reimplement the entire Windows API, which consists of thousands of complicated functions: this is almost as much work as implementing Windows itself, something which took Microsoft thousands of person-years. And if you make one tiny mistake or leave out one function that an application needs, that application will crash.

The Two Forces at Microsoft

There are two opposing forces inside Microsoft, which I will refer to, somewhat tongue-in-cheek, as *The Raymond Chen Camp* and *The* MSDN *Magazine Camp.*

Raymond Chen is a developer on the Windows team at Microsoft. He's been there since 1992, and his weblog The Old New Thing is chock-full of detailed technical stories about why certain things are the way they are in Windows, even silly things, which turn out to have very good reasons.

The most impressive things to read on Raymond's weblog are the stories of the incredible efforts the Windows team has made over the years to support backwards compatibility:

> Look at the scenario from the customer's standpoint. You bought programs X, Y and Z. You then upgraded to Windows XP. Your computer now crashes randomly, and program Z doesn't work at all. You're going to tell your friends, "Don't upgrade to Windows XP. It crashes randomly, and it's not compatible with program Z." Are you going to debug your system to determine that program X is causing the crashes, and that program Z doesn't work because it is using undocumented window messages? Of course not. You're going to return the Windows XP box for a refund. (You bought programs X, Y, and Z some months ago. The 30-day return policy no longer applies to them. The only thing you can return is Windows XP.)

I first heard about this from one of the developers of the hit game SimCity, who told me that there was a critical bug in his application: it used memory right after freeing it, a major no-no that *happened* to work OK on DOS but would not work under

Windows where memory that is freed is likely to be snatched up by another running application right away. The testers on the Windows team were going through various popular applications, testing them to make sure they worked OK, but SimCity kept crashing. They reported this to the Windows developers, who disassembled SimCity, stepped through it in a debugger, found the bug, and *added special code* that checked if SimCity was running, and if it did, *ran the memory allocator in a special mode in which you could still use memory after freeing it.*

This was not an unusual case. The Windows testing team is huge and one of their most important responsibilities is guaranteeing that everyone can safely upgrade their operating system, no matter what applications they have installed, and those applications will continue to run, even if those applications do bad things or use undocumented functions or rely on buggy behavior that happens to be buggy in Windows *n* but is no longer buggy in Windows *n*+1. In fact if you poke around in the AppCompatibility section of your registry you'll see a whole list of applications that Windows treats specially, emulating various old bugs and quirky behaviors so they'll continue to work. Raymond Chen writes, "I get particularly furious when people accuse Microsoft of maliciously breaking applications during OS upgrades. If any application failed to run on Windows 95, I took it as a personal failure. I spent many sleepless nights fixing bugs in third-party programs just so they could keep running on Windows 95."

A lot of developers and engineers don't agree with this way of working. If the application did something bad, or relied on some undocumented behavior, they think, it should just break when the OS gets upgraded. The developers of the Macintosh OS at Apple have always been in this camp. It's why so few applications from the early days of the Macintosh still work. For example, a lot of developers used to try to make their Macintosh applications run faster by copying pointers out of the jump table and calling them directly instead of using the interrupt feature of the processor like they were supposed to. Even though somewhere in *Inside Macintosh,* Apple's official Bible of Macintosh programming, there was a tech note saying "you can't do this," they did it, and it worked, and their programs ran faster . . . until the next version of the OS came out and they didn't run at all. If the company that made the application went out of business (and most of them did), well, tough luck, bubby.

To contrast, I've got DOS applications that I wrote in 1983 for the very original IBM PC that still run flawlessly, thanks to the Raymond Chen Camp at Microsoft. I know, it's not just Raymond, of course: it's the whole *modus operandi* of the core Windows API team. But Raymond has publicized it the most through his excellent website The Old New Thing so I'll name it after him.

That's one camp. The other camp is what I'm going to call the MSDN *Magazine* camp, which I will name after the developer's magazine full of exciting articles about all the different ways you can shoot yourself in the foot by using esoteric combinations of Microsoft products in your own software. The MSDN Magazine Camp is always trying to convince you to use new and complicated external technology like COM+, MSMQ, MSDE, Microsoft Office, Internet Explorer and its components, MSXML, DirectX (the very latest version, please), Windows Media Player, and Sharepoint . . . Sharepoint! which *nobody* has; a veritable panoply of *external dependencies* each one of which is going to be a huge headache when you ship your application to a paying customer and it doesn't work right. The technical name for this is DLL Hell. It works here: why doesn't it work there?

The Raymond Chen Camp believes in making things easy for developers by making it easy to write once and run anywhere (well, on any Windows box). The MSDN Magazine Camp believes in making things easy for developers by giving them really powerful chunks of code which they can leverage, if they are willing to pay the price of incredibly complicated deployment and installation headaches, not to mention the huge learning curve. The Raymond Chen camp is all about consolidation. Please, don't make things any worse, let's just keep making what we already have *still work*. The MSDN Magazine Camp needs to keep churning out new gigantic pieces of technology that nobody can keep up with.

Here's why this matters.

Microsoft Lost the Backwards Compatibility Religion

Inside Microsoft, the MSDN Magazine Camp has won the battle.

The first big win was making Visual Basic.NET not backwards-compatible with VB 6.0. This was literally the first time in living memory that when you bought an upgrade to a Microsoft product, your old *data* (i.e. the code you had written in VB6) could not be imported perfectly and silently. It was the first time a Microsoft upgrade did not respect the work that users did using the previous version of a product.

And the sky didn't *seem* to fall, not inside Microsoft. VB6 developers were up in arms, but they were disappearing anyway, because most of them were corporate developers who were migrating to web development anyway. The real long term damage was hidden.

With this major victory under their belts, the MSDN Magazine Camp took over. Suddenly it was OK to change things. IIS 6.0 came out with a different threading model that broke some old applications. I was shocked to discover that our customers with Windows Server 2003 were having trouble running FogBugz. Then .NET 1.1 was not perfectly backwards compatible with 1.0. And now that the cat was out of the bag, the OS team got into the spirit and decided that instead of adding features to the Windows API, they were going to completely replace it. Instead of Win32, we are told, we should now start getting ready for WinFX: the next generation Windows API. All different. Based on .NET with managed code. XAML. Avalon. Yes, vastly superior to Win32, I admit it. But not an upgrade: a break with the past.

Outside developers, who were never particularly happy with the complexity of Windows development, have defected from the Microsoft platform *en-masse* and are now developing for the web. Paul Graham, who created Yahoo! Stores in the early days of the dotcom boom, summarized it eloquently: "There is all the more reason for startups to write Web-based software now, because writing desktop software has become a lot less fun. If you want to write desktop software now you do it on Microsoft's terms, calling their APIs and working around their buggy OS. And if you manage to write something that takes off, you may find that you were merely doing market research for Microsoft."

Microsoft got big enough, with too many developers, and they were too addicted to upgrade revenues, so they suddenly decided that reinventing *everything* was *not* too big a project. Heck, we can do it twice. The old Microsoft, the Microsoft of Raymond Chen, might have implemented things like Avalon, the new graphics system, as a series of DLLs that can run on any version of Windows and which could be bundled with applications that need them. There's no technical reason not to do this. But Microsoft needs to give you a reason to buy Longhorn, and what they're trying to pull off is a sea change, similar to the sea change that occurred when Windows replaced DOS. The trouble is that Longhorn is not a very big advance over Windows XP; not nearly as big as Windows was over DOS. It probably won't be compelling enough to get people to buy all new computers and applications like they did for Windows. Well, maybe it will, Microsoft certainly needs it to be, but what I've seen so far is not very convincing. A lot of the bets Microsoft made are the wrong ones. For example, WinFS, advertised as a way to make searching work by making the file system be a relational database, ignores the fact that the *real* way to make searching work is *by making searching work*. Don't make me type metadata for all my files that I can search using a query language. Just do me a favor and *search the damned hard drive, quickly, for the string I typed, using full-text indexes and other technologies that were boring in 1973.*

Automatic Transmissions Win the Day

Don't get me wrong . . . I think .NET is a great development environment and Avalon with XAML is a tremendous advance over the old way of writing GUI apps for Windows. The biggest advantage of .NET is the fact that it has automatic memory management.

A lot of us thought in the 1990s that the big battle would be between procedural and object oriented programming, and we thought that object oriented programming would provide a big boost in programmer productivity. I thought that, too. Some people still think that. It turns out we were wrong. Object oriented programming is handy dandy, but it's not really the productivity booster that was promised. The *real* significant productivity advance we've had in programming has been from languages which manage memory for you automatically. It can be with reference counting or garbage collection; it can be Java, Lisp, Visual Basic (even 1.0), Smalltalk, or any of a number of scripting languages. If your programming language allows you to grab a chunk of memory without thinking about how it's going to be released when you're done with it, you're using a managed-memory language, and you are going to be much more efficient than someone using a language in which you have to explicitly manage memory. Whenever you hear someone bragging about how productive their language is, they're probably getting most of that productivity from the automated memory management, even if they misattribute it.

Racing car aficionados will probably send me hate mail for this, but my experience has been that there is only one case, in normal driving, where a good automatic transmission is inferior to a manual transmission. Similarly in software development: in almost every case, automatic memory management is superior to manual memory management and results in far greater programmer productivity.

Sidebar
Why does automatic memory management make you so much more productive?

1) Because you can write *f(g(x))* without worrying about how to free the return value from g, which means you can use functions which return interesting complex data types and functions which transform interesting complex data types, in turn allowing you to work at a higher level of abstraction.

2) Because you don't have to spend any time writing code to free memory or tracking down memory leaks.

3) Because you don't have to carefully coordinate the exit points from your functions to make sure things are cleaned up properly.

If you were developing desktop applications in the early years of Windows, Microsoft offered you two ways to do it: writing C code which calls the Windows API directly and managing your own memory, or using Visual Basic and getting your memory managed for you. These are the two development environments I have used the most, personally, over the last 13 years or so, and I know them inside-out, and my experience has been that Visual Basic is *significantly* more productive. Often I've written *the same code,* once in C++ calling the Windows API and once in Visual Basic, and C++ always took three or four times as much work. Why? Memory management. The easiest way to see why is to look at the documentation for any Windows API function that needs to return a string. Look closely at how much discussion there is around the concept of who allocates the memory for the string, and how you negotiate how much memory will be needed. Typically, you have to call the function *twice*—on the first call, you tell it that you've allocated zero bytes, and it fails with a "not enough memory allocated" message and conveniently also tells you how much memory you need to allocate. That's if you're lucky enough not to be calling a function which returns a *list of strings* or a whole variable-length structure. In any case, simple operations like opening a file, writing a string, and closing it using the raw Windows API can take a page of code. In Visual Basic similar operations can take three lines.

So, you've got these two programming worlds. Everyone has pretty much decided that the world of managed code is far superior to the world of unmanaged code. Visual Basic was (and probably remains) the number one bestselling language product of all time and developers preferred it over C or C++ for Windows development, although the fact that "Basic" was in the name of the product made hardcore programmers shun it even though it was a fairly modern language with a handful of object-oriented features and very little leftover gunk (line numbers and the LET statement having gone the way of the hula hoop). The other problem with VB was that deployment required shipping a VB runtime, which was a big deal for shareware distributed over modems, and, worse, let other programmers see that your application was developed in (*the shame*!) Visual

Basic.

One Runtime To Rule Them All

And along came .NET. This was a grand project, the super-duper unifying project to clean up the whole mess once and for all. It would have memory management, of course. It would still have Visual Basic, but it would gain a new language, one which is in spirit virtually the same as Visual Basic but with the C-like syntax of curly braces and semicolons. And best of all, the new Visual Basic/C hybrid would be called Visual C#, so you would not have to tell anyone you were a "Basic" programmer any more. All those horrid Windows functions with their tails and hooks and backwards-compatibility bugs and impossible-to-figure-out string-returning semantics would be wiped out, replaced by a single clean object oriented interface that only has one kind of string. One runtime to rule them all. It was beautiful. And they pulled it off, technically. .NET is a great programming environment that manages your memory and has a rich, complete, and consistent interface to the operating system and a rich, super complete, and elegant object library for basic operations.

And yet, people aren't really using .NET much.

Oh sure, some of them are.

But the idea of unifying the mess of Visual Basic and Windows API programming by creating a *completely new, ground-up* programming environment with not one, not two, but three languages (or are there four?) is sort of like the idea of getting two quarreling kids to stop arguing by shouting "shut up!" louder than either of them. It only works on TV. In real life when you shout "shut up!" to two people arguing loudly you just create a louder three-way argument.

(By the way, for those of you who follow the arcane but politically-charged world of blog syndication feed formats, you can see the same thing happening over there. RSS became fragmented with several different versions, inaccurate specs and lots of political fighting, and the attempt to clean everything up by creating *yet another format* called Atom has resulted in several different versions of RSS plus one version of Atom, inaccurate specs and lots of political fighting. When you try to unify two opposing forces by creating a third alternative, you just end up with three opposing forces. You haven't unified anything and you haven't really fixed anything.)

So now instead of .NET unifying and simplifying, we have a big 6-way mess, with everybody trying to figure out which development strategy to use and whether they can afford to port their existing applications to .NET.

No matter how consistent Microsoft is in their marketing message ("just use .NET—trust us!"), most of their customers are still using C, C++, Visual Basic 6.0, and classic ASP, not to mention all the other development tools from other companies. And the ones that are using .NET are using ASP.NET to develop web applications, which run on a Windows server but *don't require Windows clients*, which is a key point I'll talk about more when I talk about the web.

Oh, Wait, There's More Coming!

Now Microsoft has so many developers cranking away that it's not enough to reinvent the entire Windows API: they have to reinvent it *twice*. At last year's PDC they preannounced the next major version of their operating system, codenamed *Longhorn*, which will contain, among other things, a completely new user interface API, codenamed *Avalon*, rebuilt from the ground up to take advantage of modern computers' fast display adapters and realtime 3D rendering. And if you're developing a Windows GUI app today using Microsoft's "official" latest-and-greatest Windows programming environment, WinForms, you're going to have to start over again in two years to support Longhorn and Avalon. Which explains why WinForms is completely stillborn. Hope you haven't invested too much in it. Jon Udell found a slide from Microsoft labelled "How Do I Pick Between Windows Forms and Avalon?" and asks, "Why do I have to pick between Windows Forms and Avalon?" A good question, and one to which he finds no great answer.

So you've got the Windows API, you've got VB, and now you've got .NET, in several language flavors, and don't get too attached to any of that, because we're making Avalon, you see, which will only run on the newest Microsoft operating system, which nobody will have for a loooong time. And personally I still haven't had time to learn .NET very deeply, and we haven't ported Fog Creek's two applications from classic ASP and Visual Basic 6.0 to .NET because *there's no return on investment for us. None.* It's just Fire and Motion as far as I'm concerned: Microsoft would love for me to stop adding new features to our bug tracking software and content management software and instead waste a few months porting it to another programming environment, something which will not benefit a single customer and therefore will not gain us one additional sale, and therefore which is a complete waste of several months, which is great for Microsoft, because they have content management software and bug tracking software, too, so they'd like nothing better than for me to waste time spinning cycles catching up with the flavor du jour, and then waste another year or two doing an Avalon version, too, while they add features to their own competitive software. *Riiiight.*

No developer with a day job has time to keep up with all the new development tools coming out of Redmond, if only because there are *too many dang employees at Microsoft making development tools!*

It's Not 1990

Microsoft grew up during the 1980s and 1990s, when the growth in personal computers was so dramatic that every year there were more new computers sold than the entire installed base. That meant that if you made a product that only worked on new computers, within a year or two it could take over the world even if nobody *switched* to your product. That was one of the reasons Word and Excel displaced WordPerfect and Lotus so thoroughly: Microsoft just waited for the next big wave of hardware upgrades and sold Windows, Word and Excel to corporations buying their next round of desktop computers (in some cases their first round). So in many ways Microsoft never needed to learn how to get an installed base to switch from product N to product N+1. When people get new computers, they're happy to get all the latest Microsoft stuff on the new computer, but they're far less likely to upgrade. This didn't matter when the PC industry was growing like wildfire, but now that the world is saturated with PCs most of which are Just Fine, Thank You, Microsoft is suddenly realizing that it takes much longer for the latest thing to get out there. When they tried to "End Of Life" Windows 98, it turned out there were still so many people using it they had to promise to support that old creaking grandma for a few more years.

Unfortunately, these Brave New Strategies, things like .NET and Longhorn and Avalon, trying to create a *new* API to lock people into, can't work very well if everybody is still using their good-enough computers from 1998. Even if Longhorn ships when it's supposed to, in 2006, which I don't believe for a minute, it will take a couple of years before enough people have it that it's even worth considering as a development platform. Developers, developers, developers, and developers are not buying into Microsoft's multiple-personality-disordered suggestions for how we should develop software.

Enter the Web

I'm not sure how I managed to get this far without mentioning the Web. Every developer has a choice to make when they plan a new software application: they can build it for the web or they can build a "rich client" application that runs on PCs. The basic pros and cons are simple: Web applications are easier to deploy, while rich clients offer faster response time enabling much more interesting user interfaces.

Web Applications are easier to deploy because there's no installation involved. Installing a web application means typing a URL in the address bar. Today I installed Google's new email application by typing Alt+D, gmail, Ctrl+Enter. There are far fewer compatibility problems and problems coexisting with other software. Every user of your product is using the same version so you never have to support a mix of old versions. You can use any programming environment you want because you only have to get it up and running on your own server. Your application is automatically available at virtually every reasonable computer *on the planet*. Your customers' data, too, is automatically available at virtually every reasonable computer on the planet.

But there's a price to pay in the smoothness of the user interface. Here are a few examples of things you can't really do well in a web application:

1. Create a fast drawing program
2. Build a real-time spell checker with wavy red underlines
3. Warn users that they are going to lose their work if they hit the close box of the browser
4. Update a small part of the display based on a change that the user makes without a full roundtrip to the server
5. Create a fast keyboard-driven interface that doesn't require the mouse
6. Let people continue working when they are not connected to the Internet

These are not all big issues. Some of them will be solved very soon by witty Javascript developers. Two new web applications, Gmail and Oddpost, both email apps, do a really decent job of working around or completely solving some of these issues. And users don't seem to care about the little UI glitches and slowness of web interfaces. Almost all the normal people I know are perfectly happy with web-based email, for some reason, no matter how much I try to convince them that the rich client is, uh, *richer.*

So the Web user interface is about 80% there, and even without new web browsers we can probably get 95% there. This is Good Enough for most people and it's certainly good enough for developers, who have voted to develop almost every significant new application as a web application.

Which means, suddenly, Microsoft's API doesn't matter so much. *Web applications don't require Windows.*

It's not that Microsoft didn't notice this was happening. Of course they did, and when the implications became clear, they slammed on the brakes. Promising new technologies like HTAs and DHTML were stopped in their tracks. The Internet Explorer team seems to have disappeared; they have been completely missing in action for several years. There's no way Microsoft is going to allow DHTML to get any better than it already is: it's just too dangerous to their core business, the rich client. The big meme at Microsoft these days is: "*Microsoft is betting the company on the rich client*." You'll see that somewhere in every slide presentation about Longhorn. Joe Beda, from the Avalon team, says that "Avalon, and Longhorn in general, is Microsoft's stake in the ground, saying that we believe power on your desktop, locally sitting there doing cool stuff, is here to stay. We're investing on the desktop, we think it's a good place to be, and we hope we're going to start a wave of excitement . . . "

The trouble is: it's too late.

I'm a Little Bit Sad About This, Myself

I'm actually a little bit sad about this, myself. To me the Web is great but Web-based applications with their sucky, high-latency, inconsistent user interfaces are a huge step backwards in daily usability. I love my rich client applications and would go nuts if I had to use web versions of the applications I use daily: Visual Studio, CityDesk, Outlook, Corel PhotoPaint, QuickBooks. But that's what developers are going to give us. Nobody (by which, again, I mean "fewer than 10,000,000 people") wants to develop for the Windows API any more. Venture Capitalists won't invest in Windows applications because they're so afraid of competition from Microsoft. And most users don't seem to care about crappy Web UIs as much as I do.

And here's the clincher: I noticed (and confirmed this with a recruiter friend) that Windows API programmers here in New York City who know C++ and COM programming earn about $130,000 a year, while typical Web programmers using managed code languages (Java, PHP, Perl, even ASP.NET) earn about $80,000 a year. That's a huge difference, and when I talked to some friends from Microsoft Consulting Services about this they admitted that Microsoft had lost a whole generation of developers. The reason it takes $130,000 to hire someone with COM experience is because nobody bothered learning COM programming in the last eight years or so, so you have to find somebody really senior, usually they're already in management, and convince them to take a job as a grunt programmer, dealing with (God help me) *marshalling* and monikers and apartment threading and aggregates and tearoffs and a million other things that, basically, only Don Box ever understood, and even Don Box can't bear to look at them any more.

Much as I hate to say it, a huge chunk of developers have long since moved to the web and refuse to move back. Most .NET developers are ASP.NET developers, developing for Microsoft's web server. ASP.NET is brilliant; I've been working with web development for ten years and it's really just a generation ahead of everything out there. But it's a server technology, so clients can use any kind of desktop they want. And it runs pretty well under Linux using Mono.

None of this bodes well for Microsoft and the profits it enjoyed thanks to its API power. The new API is HTML, and the new winners in the application development marketplace will be the people who can make HTML sing.