Does Visual Studio Rot the Mind?

Ruminations on the Psychology and Aesthetics of Coding

By Charles Petzold

A Talk Delivered at the NYC .NET Developer's Group, October 20, 2005

Abstract: Visual Studio can be one of the programmer's best friends, but over the years it has become increasingly pushy, domineering, and suffering from unsettling control issues. Should we just surrender to Visual Studio's insistence on writing our code for us? Or is Visual Studio sapping our programming intelligence rather than augmenting it? This talk dissects the code generated by Visual Studio; analyzes the appalling programming practices it perpetuates; rhapsodizes about the joys, frustrations, and satisfactions of unassisted coding; and speculates about the radical changes that Avalon will bring.

I don't speak to user groups or developer groups very often, partially because of a bad experience I had a number of years ago. In 1991, the Boston Computer Society invited me to speak on the subject of Windows programming. Since I had written a book on Windows programming — the 2nd edition of *Programming Windows* had been published just the previous year — I figured I didn't really need to talk about that. Anybody who wanted to learn Windows programming could simply buy the book and read it in the privacy of their homes.

Instead, I decided to talk about something that people could not read about in my book. This was a subject that interested me at the time, which was how Microsoft first developed Windows, and how Microsoft and IBM then got involved in the development of OS/2, how Microsoft had convinced IBM to go graphical in the windowing environment, and how IBM decided they wanted to develop an entirely new API, and then how the success of Windows led to the eventual split between Microsoft and IBM.

Well, people went berserk. I suppose they had been expecting me to tell them everything they needed to know about Windows programming in the space of a few hours, and that's certainly not what I was interested in doing.

That was a very traumatic experience for me, and since that time, I have been reticent about speaking before groups like this. This is a .NET developer's group. I have written four books on .NET programming — the fourth is due out on November 2nd — and anybody who wants to read them is free to do so. I am now in the process of writing a book on the Windows Presentation Foundation (formerly code-named Avalon) but couldn't at this time really tell you a whole lot that's useful.

So when Bill and Andrew and Stephen asked me to speak here, I was initially reticent until I was

told — and I quote from Andrew's email — "we'd happily go to great lengths to accommodate you and your topic preference, even if your preference is no particular topic at all." This evening I do have a topic, although it's a rather odd one, and I hope nobody will go berserk.

Computers in the Movies

I recently re-watched what I believe to be the first Hollywood movie to deal with computers in a major way. This is the movie *Desk Set*. It's 1957, and Katherine Hepburn supervises three other women in the research department of the Federal Broadcasting Company. Whenever somebody has a question requiring research, they call up on the phone, and Hepburn and her team get to work on the answer.

Enter Spencer Tracy, a self-described Methods Engineer and the inventor of an "electronic brain" called the Electro-Magnetic Memory and Research Arithmetical Calculator or EMERAC —"Emmy" for short.

Well, naturally, everybody in the research department assumes they're going to be fired, just like what happened in payroll when an "electronic brain" was installed there. Although it's expressed here as a nervous humor, this plot line reflected a real anxiety at the time. Of course, workers didn't really think that an entire computer was necessary to do their job. No, people instead joked that they were going to be replaced by a single *button*.

Emmy the "electronic brain" turns out to be quite a sophisticated computer. It understands natural language questions typed from a keyboard, and prints the answers on a teletypewriter. Of course, the computer can't match the cool professionalism of Katherine Hepburn and her staff, and when confronted with a tough question, it reacts in ways we're all familiar with from subsequent movie computers: It makes funny noises, smoke starts pouring out, and it begins spitting Hollerith cards all over the office.

It must have been funny at the time, even for people who knew that computers don't break down that way in real life. I have been programming pretty steadily for about 30 years now, and only once have I seen a computer fail in a way I'd classify as "interesting."[1]

Meanwhile, the "electronic brain" installed in payroll starts printing pink slips for everybody in the company. As we all know, once computers take over, they no longer need humans. But, as you may anticipate, the human staff is then allowed to come to the rescue and display its superiority over the machine.

Because *Desk Set* is a Hollywood romantic comedy, and perhaps because IBM supplied all the hardware for the film, human and machine are reconciled at the end. The computer's role is clarified. It exists merely to *assist* the humans in their research needs.

The idea of computers trying to "take over" their human creators has been replayed in dozens of movies ever since. Sometimes the computers take a leap into a level of intelligence and consciousness their designers had not anticipated, or perhaps they suffer the computer version of a "nervous breakdown," or they're simply too damn *logical* for their own good. Sometimes the computer goes so haywire that it does something that no sane human would ever dream of doing — such as starting a war.

Just a few examples[2]: *2001: A Space Odyssey* (1968). Good computer; bad choices. *Colossus: the Forbin Project* (1970). American super-computer demands to be connected to Russian super-

computer, with expected consequences. As the movie's tag line put it, "We built a super computer with a mind of its own and now we must fight it for the world!"[3]

Ira Levin's novel *This Perfect Day* from 1970 isn't a movie of course, but it should be. A computer runs the world and a few brave souls conspire to defeat it, only to discover… well, this is an Ira Levin novel so it's best that you be a little surprised. *This Perfect Day*. Check it out.

Many of the human vs. computer themes show up in *WarGames* (1983), and even this past summer, Hollywood squeezed out a little cinematic turd called *Stealth*, which my favorite newspaper *The Onion* hailed as "the most chilling look at artificial intelligence run amok since *Short Circuit 2*."[4]

Of course, it's easy today to see how ridiculous this stuff is. For example, we have erected a vast network of computers around the world, and it has yet to exhibit the slightest degree of intrinsic intelligence. It's fairly easy to imagine the whole thing collapsing, or getting infected with some virus, but not so easy to imagine it deviously turning against us. What the Internet seems to do best is make commonly available enormously vast resources of *mis*-information that we never knew existed.

In thinking about how computers and people interact, I find some older, pre-computer, movies to be more relevant — the ones where people who work with machines and automated processes become subservient to the *rhythm* of the machine. I'm thinking of Fritz Lang's *Metropolis* (1927), where an odd feedback system requires people to run the machines by responding to their commands. The opening scenes of Charlie Chaplin's *Modern Times* (1936) are still very funny, and let's not forget the famous episode of *I Love Lucy* called "Job Switching" (1952), the one where Lucy gets a job at the candy factory.

Although these scenes are often played for laughs, they seem to reflect our fear of being forced into a dehumanizing relationship with a machine that imposes its jerky digital rhythm on our normally analogue souls. These encounters leave an imprint on both minds and our bodies. In *I Am a Fugitive from a Slave Gang* (1932), a convict who is set free from the chain gang walks away still shuffling his feet as if his ankles are still bound with the chains. Similarly, four years later in *Modern Times*, Chaplin walks home from work with his arms still twitching in the rhythm of the machine.

Technological Addictions

When using our computers, something a little different happens. Of course, we may get repetitive stress disorders from the keyboard or mouse, but software doesn't affect our bodies as much as our minds, and even then in subtle ways.

Some observers of our digital lives have noticed the way in which certain applications cause a user to think in very rigid prescribed ways, and these are not good. One of the biggest offenders, of course, is PowerPoint. Start putting what you want to communicate in PowerPoint slides, and everything you want to say is ordered into half a dozen bullet items.

The critiques of technology we see in the movies seem to use metaphors of power or slavery. I think there's a more proper metaphor for our relationships with much of modern consumer technology, however, and that metaphor is *addiction*.

It is very common for us to say about a piece of consumer technology that "we didn't know how much we needed it until we had it," and much of this technology seems targeted not to satisfy a

particular need, but to get us hooked on something else we never knew we needed; not to make our lives better, but to tempt us with another designer drug. “I can’t live without my _____” and you can fill in the blank. This week, I think, it’s the video iPod.

Even useful items like cell phones seem to fit this model. Was the cell phone wasn’t invented to fill a particular need? I’m not sure. It probably wasn’t the case of someone sitting on M14 bus thinking “I’m bored. I wish I could call up my friends from this bus.” It was partially invented because it was *possible* to do it, and there was a trust that people eventually would not be able to live without them. Once we get the taste, we’re hooked.

I often wish that email had never been invented, but there’s just no way I can get rid of it. So, day after day, several times a day, I dutifully delete 99% of the emails I receive, and when I’m not able to get at my email for a few days, I’ll leave the machine at home running to pick it up every 10 minutes so I don’t overflow some capacity somewhere, and just the other day I caught myself wondering who will clean out my Inbox after I’m dead.

We see computer technologies going through a life cycle where they turn from beneficial to evil. I recently installed something from a Microsoft site that dutifully gave me instructions about clicking the yellow bar at the top of Internet Explorer — the yellow bar that is now protecting us from popups and other bad stuff coming through — so I could bypass the normal protection against the installation of Active X controls. The instructions go on to tell you to ignore all the warnings Windows will then provide about installing this Active X control, because this particular one, unlike all the others, actually does a good thing.

The Nuisance of Information at Your Fingertips

Sometime in the early 1990s, Microsoft came up with the slogan “Information at your Fingertips.” It sounds pretty good in theory, and today we basically have a whole lot of information at our fingertips, what with Google and Wikipedia and the Internet Movie Database and other useful sites.

But information too has become an addiction. I don’t know what it’s like at your house, but we can’t even watch a simple rerun of *Friends* without one of us saying “Who’s that actress playing Joey’s girlfriend? She looks familiar.” We then click the Info button on the remote to get the title of the particular episode, and then go to tvtome.com or epguides.com to learn the identity of the actress, and then perhaps to Internet Movie Database to see a list of the movies, TV shows, and guest appearances she’s done, and perhaps for one of the movies we’ll say “I always wanted to see that movie” and so we add it to our NetFlix queue.

Of course, we can do all this without actually leaving the couch. Why else would we need WiFi in a 400 square foot studio apartment?

Now I know that five or ten years from now we’ll be able to perform this entire operation entirely from the cable remote, which may actually be a computer remote, including pausing the episode of *Friends* to download the movie in which the actress playing Joey’s girlfriend appears, and watch it on demand, and then go back to the episode of *Friends* we were watching if we so desire. And I might be more thrilled at this prospect if I thought it would make us better, happier, nicer human beings. But that’s not immediately obvious to me.

And I sincerely hope that in five or ten years from now, we’re not still watching reruns of *Friends*.

Although we here in this room are computer users and we are thus stuck with the same annoyances,

distractions, and addictions as all computer users, we are also developers and programmers, and for us, the computing experience *should* be intrinsically different. Programming is the most empowering thing we can do on a computer, and that's what we do. We write the code that makes the whole world sing.

And yet, if all of us in this room are .NET developers, then we are also undoubtedly Visual Studio users. I suspect some of us have been playing around with betas of Visual Studio 2005, and awaiting its official release at DevConnections in Las Vegas in a few weeks.

Life without Visual Studio is unimaginable, and yet, no less than PowerPoint, Visual Studio causes us to do our jobs in various predefined ways, and I, for one, would be much happier if Visual Studio did much less than what it does. Certain features in Visual Studio are supposed to make us more productive, and yet for me, they seem to denigrate and degrade the programming experience.

API Proliferation

Twenty years ago, in November 1985, Windows 1.0 debuted with approximately 400 documented function calls.[5] Ten years later, Windows 95 had well over a thousand.[6]

Today we are ready for the official release of the .NET Framework 2.0. Tabulating only MSCORLIB.DLL and those assemblies that begin with word System, we have over 5,000 public classes that include over 45,000 public methods and 15,000 public properties, not counting those methods and properties that are inherited and not overridden. A book that simply listed the names, return values, and arguments of these methods and properties, one per line, would be about a thousand pages long.

If you wrote each of those 60,000 properties and methods on a 3-by-5 index card with a little description of what it did, you'd have a stack that totaled 40 feet.[7] These 60,000 cards, laid out end to end — the five inch end, not the three inch end — can encircle Central Park (almost), and I hear this will actually be a public art project next summer.

Can any one programmer master 60,000 methods and properties? I think not. One solution, of course, is specialization. I myself have specialized. This evening I hope no one will ask me questions about web forms or ASP .NET or SQL Server because those aren't my specialty. I do Windows Forms, and my language is C#.

IntelliSense

Visual Studio has attempted to alleviate the problem of class, method, and property proliferation with a feature called IntelliSense. IntelliSense indeed puts information at our fingertips, if you think of your fingertips figuratively as that place on the screen where the keyboard caret is.

Like other addictive technologies, I have a love/hate relationship with IntelliSense, and the more I despise it, the more I use it, and the more I use it, the more disgusted I am at how addicted I've gotten, and the more addicted I get, the more I wish it had never been invented.

Just in case you've been out of the trenches for awhile, IntelliSense is a culmination of some past attempts at code completion technologies. If you type an object name and a period, for example, you'll get a little scrollable dropdown menu with a list of all the public methods, properties, and events for that class, and when you choose a method name and type a left parenthesis, you'll get the

various overloads with arguments, and a little tooltip describing what the method does.

IntelliSense is considered by some to be the most important programming innovation since caffeine. It works especially well with .NET because Visual Studio can use reflection to obtain all the information it needs from the actual DLLs you've specified as references.

In fact, IntelliSense has become the first warning sign that you haven't properly included a DLL reference or a *using* directive at the top of your code. You start typing and IntelliSense comes up with nothing. You know immediately something is wrong.

And yet, IntelliSense is also dictating the way we program.

For example, for many years programmers have debated whether it's best to code in a top-down manner, where you basically start with the overall structure of the program and then eventually code the more detailed routines at the bottom; or, alternatively, the bottom-up approach, where you start with the low-level functions and then proceed upwards. Some languages, such as classical Pascal, basically impose a bottom-up approach, but other languages do not.

Well, the debate is now over. In order to get IntelliSense to work correctly, bottom-up programming is best. IntelliSense wants every class, every method, every property, every field, every method parameter, every local variable properly defined before you refer to it. If that's not the case, then IntelliSense will try to correct what you're typing by using something that *has* been defined, and which is probably just plain wrong.

For example, suppose you're typing some code and you decide you need a variable named *id*, and instead of defining it first, you start typing a statement that begins with *id* and a space. I always type a space between my variable and the equals sign. Because *id* is not defined anywhere, IntelliSense will find something that begins with those two letters that is syntactically correct in accordance with the references, namespaces, and context of your code. In my particular case, IntelliSense decided that I really wanted to define a variable of interface type *IDataGridColumnStyleEditingNotificationService*, an interface I've never had occasion to use.

On the plus side, if you really need to define an object of type *IDataGridColumnStyleEditingNotificationService*, all you need do is type *id* and a space.

If that's wrong, you can eliminate IntelliSense's proposed code and go back to what you originally typed with the Undo key combination Ctrl-Z. I wish I could slap its hand and say "No," but Ctrl-Z is the only thing that works. Who could ever have guess that Ctrl-Z would become one of the most important keystrokes in using modern Windows applications? Ctrl-Z works in Microsoft Word as well, when Word is overly aggressive about fixing your typing.

But the implication here is staggering. To get IntelliSense to work right, not only must you code in a bottom-up structure, but within each method or property, you must also write you code linearly from beginning to end — just as if you were using that old DOS line editor, EDLIN. You must define all variables before you use them. No more skipping around in your code.

It's not that IntelliSense is teaching us to program like a machine; it's just that IntelliSense would be much happier if we did.

And I think it's making us dumber. Instead of finding out exactly the method I need, or instead of trying to remember an elusive property name, I find myself scrolling through the possibilities that IntelliSense provides, looking for a familiar name, or at least something that seems like it might do

the job.

I don't need to remember *anything* any more. IntelliSense will remember it for me. Besides, I justify to myself, I may not want those 60,000 methods and properties cluttering up my mind. My overall mental health will undoubtedly be better without them, but at the same time I'm prevented from ever achieving a fluid coding style because the coding is not coming entirely from my head. My coding has become a constant dialog with IntelliSense.

So I don't think IntelliSense is helping us become better programmers. The real objective is for us to become *faster* programmers, which also means that it's cheapening our labor.

Of course, I could always just turn it off.

And go cold turkey? I don't think so!

IntelliSense is a technology that is inevitable. If Microsoft hadn't done it, somebody else would have. Human beings have never been inclined to refrain from pursuing certain technologies because they may have unfortunate repercussions.

Generated Code

Not only does Visual Studio try to complete code we're typing, but for many years now, Visual Studio has wanted to generate code for us. If we select a new project type of Windows Application, for example, and give it a name and location on a local drive, Visual Studio generates sufficient code so that this project is immediately compilable and runable.

Somehow, we have been persuaded that this is the proper way to program. I don't know why. Personally, I find starting a program with an empty source code file to be very enjoyable. I like typing the preliminaries and then the *main* function or the *Main* method. The time when I can really use some help is *not* when I'm starting a program, but when I'm trying to *finish* it. Where is Visual Studio then?

Let's go into Visual Studio 2005 in our imaginations and create a C# Windows Forms program — a project of type Windows Application — and let's examine the code.

First, we see that Visual Studio has hooked up our program to a bunch of dynamic link libraries it thinks we will need. It has created references not only to the System, System.Drawing, and System.Windows.Forms assemblies that are required of any non-trivial Windows Forms application, but also System.Data, System.Deployment, and System.Xml, regardless whether your program needs these DLLs or not. These extra references don't do any real harm except if someone else examines the program — perhaps to make some changes after the original programmer has moved on — and assumes that the application *requires* these references. Now there's confusion, because references to DLLs exist that the program really doesn't need.

Among other files, Visual Studio creates a file named Form1.cs that it graciously allows you the programmer to modify. This is the source code file where the event handlers go for controls on the main form. We see a bunch of *using* directives at the top with the namespaces the program needs. The normal namespaces for Windows Forms programs are *System*, *System.Drawing*, and *System.Windows.Forms*, but Visual Studio also includes *System.Collections.Generic*, *System.ComponentModel*, *System.Data*, and *System.Text*, some of which are useful to be sure, but if the program doesn't actually use any of the classes in these namespaces, they're just distracting

noise.

Visual Studio also encloses all the code it generates in a namespace that is given the same name as the project name. Of course, I understand entirely the value of having a namespace for DLLs, but why have a namespace for applications? I have given this much thought, and I can't figure it out.

I've been talking about code, but when you create a new Windows Forms project, Visual Studio doesn't put any of the source code files in front of your face. Instead, it puts a design view of a form which you are then expected to start interactively populating with controls.

Visual Studio also creates a file named Form1.Designer.cs, and in some beta versions of Visual Studio 2005, this file wasn't even listed among the project files by default. This is the file in which Visual Studio inserts generated code when you design your form. Visual Studio really doesn't want you messing around with this file, and for good reason. Visual Studio is expecting this generated code to be in a certain format, and if you mess with it, it may not be able to read it back in the next time you open the project.

The Origins of the Designer

Of course, the interactive design of forms is one of the major features of Visual Studio. It may help to contrast the Visual Studio designer with the way Windows programming was done 20 years ago when Windows 1.0 was first released — 20 years ago next month, to be precise.

I don't want to bore everybody with war stories about Windows 1.0 programming, but basically you did most of your programming outside of Windows on the DOS command line. You edited your source code with your favorite editor — mine was a word processor of the era called WordStar — and then you'd compile and link on the command line, and run your application in Windows by typing Win followed by the name of your EXE file. You'd test out your program in Windows and then exit Windows back to DOS and reload the source code in your editor.

It was not possible to do much program development inside Windows except to run the Windows-based Icon Editor that helped you make icons, cursors, and small bitmaps.

To define the layout of menus and dialog boxes, you created a text resource script. In this file you'd type templates for the menus and dialog boxes in a particular defined format. For dialog boxes, you'd basically list the controls you wanted in the dialog box. The syntax wasn't too bad, for example, the keyword BUTTON followed by the text that appeared in the button, the button's location in the dialog box, and its width and height. Because Windows could run under systems of different display resolutions, these sizes and locations were in a special dialog box coordinate system, where horizontal units were equivalent to ¼ the width of the default system font, and vertical units were 1/8 the height of the font.

These resource scripts were actually considered an enormous time saver. If you didn't have the resource script, you'd have to create the menu and dialog boxes in code, and that was a particularly onerous job. And the syntax of the resource script was so clean and simple that it remains to this day the most concise way of defining menus and dialog boxes that I have ever seen.

There was only one little problem: All those controls in your dialog boxes had to be positioned and sized correctly with numbers you had to figure out. Sometimes it would help to start with a piece of graph paper and do a hand mockup of the dialog box. Otherwise, it was pretty much a matter of trial and error, back and forth, load up Windows, invoke the dialog box, see how crappy it looked,

exit Windows, reload the resource script, and eventually you'd fine tune those sizes and locations until it looked right.

Sometime in 1986 or so, beta versions of a Dialog Editor emerged from Microsoft. I found out about it by lurking on a bulletin board called the Windows Developers Roundtable on the information service GEnie, the General Electric Network for Information Exchange. The Dialog Editor was a Windows program that let you interactively design your dialog boxes by moving controls around, and would then output the dialog template in the correct format. I eventually wrote an article about it for the very first issue of *Microsoft Systems Journal* in October 1986. The article had the title "Latest Dialog Editor Speeds Windows Application Development".

But for the Windows programs that I later wrote for *MSJ*, and for those in the first edition of *Programming Windows*, I couldn't use this Dialog Editor. I couldn't use it for the simple reason that the output of the Dialog Editor was *ugly*.

Let me explain what I mean by that, and why it's a problem.

People who write code for publication have some extra burdens that most programmers need not worry about. The code we write has to be concise but not to the point where it's obscure. It should be nicely formatted and easy to read. There should be plenty of white space. It should be airy. It can't have superfluous variables or unused functions. It has to compile without warning messages.

Moreover, code published in books is limited to line lengths of about 80 characters, sometimes a little more, sometimes less. Magazines sometimes impose even more severe limits, depending how much power the art department has in the political structure of the magazine staff. *Microsoft Systems Journal* was an interesting case. Although *M SJ* was originally supposed to be strictly a Windows-programming magazine — they later chickened out and made it for DOS programming as well — it was laid out entirely on Macs and the art department ruled. Art people love very narrow text columns, and I remember *MSJ* going to extra-narrow columns at about the same time that the OS/2 Presentation Manager was introducing function calls like *GpiQueryModelTransformMatrix*.

Under these criteria, the Dialog Editor generated ugly and unusable output. It didn't even generate BUTTON, LABEL, and LISTBOX statements like a normal human being. Instead, it spit out everything as general-purpose CONTROL statements followed by class names. These lines had lots of superfluous information. All the parameters were included, although many of them were default values. These lines were way over 80 characters.

People have to read this stuff, and the Dialog Editor output was unreadable. Moreover, if somebody is trying to reproduce the dialog box on their own, it doesn't make any sense to show the output from the Dialog Editor.

When I was working on the first edition of *Programming Windows* in 1987, sometimes I'd use the Dialog Editor and then manually convert the output to something a human being might write. But mostly I just got skilled at designing dialog boxes by hand.

Of course, a Dialog Editor is certainly a piece of "inevitable technology." It's so obvious that it must be created, and in principle, I can't summon up any arguments against it. It's what it evolved into that really bothers me.

Interactive Design

Most of the really innovative interactive design stuff found its first expressions in the Windows-based versions of Visual Basic, and here's where I started becoming nervous about where Windows programming was headed. Not only could you move a button onto your form, and interactively position and size it just the way you wanted, but if you clicked on the button, Visual Basic would generate an event handler for you and let you type in the code.

This bothered me because Visual Basic was treating a program not as a complete coherent document, but as little snippets of code attached to visual objects. That's not what a program is. That's not what the compiler sees. How did one then get a sense of the complete program? It baffled me.

Eventually, the interactive design stuff found its way into development with C++ and the Microsoft Foundation Classes, and there, I truly believe, code generation was used to hide a lot of really hairy MFC support that *nobody* wanted to talk about.

For an author who writes programming books, all this stuff presents a quandary. How do you write a programming tutorial? Do you focus on using Visual Studio to develop applications? Frankly, I found it very hard to write sentences like "Now drag the button object from the tool box to your dialog box" and still feel like I was teaching programming. I never wrote about C++ and MFC, partially because MFC seemed like a light wrapper on the Windows API and barely object oriented at all. I continued to revise later editions of *Programming Windows* under the assumption that its readers were programmers like me who preferred to write their own code from scratch.

Bye, Bye, Resource Script

I first started looking at beta .NET 1.0 and Windows Forms in the summer of 2000, and it was clearly more object oriented than MFC could ever be. I liked it. I was also intrigued that the resource script had entirely disappeared. You created and assembled controls on dialog boxes — now subsumed under the more global term of "forms" — right there in your code. Of course, even in Windows 1.0, you could create controls in code, and the first edition of *Programming Windows* has some examples. But it just wasn't very pleasant, because every control creation involved a call to the 11-argument *CreateWindow* function. But creating controls in code in Windows Forms was a snap.

What's nice about creating and manipulating controls in code is that you're creating and manipulating them in code. Suppose you want a column of ten equally-spaced buttons of the same size displaying *Color* names. In a program, you can actually store those ten *Color* values in one place. It's called an *array*. There's also an excellent way to create and position these ten buttons. It's called a *for* loop. This is programming.

But Visual Studio doesn't want you to use arrays or loops to create and position these buttons. It wants you to use the designer, and it wants to generate the code for you and hide it away where you can't see it.

Almost twenty years after the first Dialog Editor, Visual Studio is now the culprit that generates ugly code and warns you not to mess with it.

If you do try to read this code, it's not that easy because every class and structure is preceded by a full namespace:

```
System.Windows.Forms.Button button1 = new System.Windows.Forms.Button();
```

Of course, we know why Visual Studio needs to do this. In theory, it's possible for you to add to the Visual Studio toolbox another control named *Button* from another DLL with another namespace, and there must be a way to differentiate them.

As Visual Studio is generating code based on the controls you select, it gives the controls standard names, such as *button1*, *button2*, *button3*, *label1*, *label2*, *label3*, *textBox1*, *textBox2*, *textBox3*.

Of course, Visual Studio lets you change that variable name. You change the *Name* property of the control, and that becomes not only the *Name* property of the button object, but also the button variable name.

Do programmers actually do this? I'm sure some do, but I'm also sure many do not. How do I know? Take a look at some of the sample code that comes out of Microsoft. There you'll see *button1*, *button2*, *button3*, *label1*, *label2*, *label3*, etc. Any yet, everyone agrees that one of the most important elements of writing self-documenting code is giving your variables and objects meaningful names.

If Visual Studio really wanted you to write good code, every time you dragged a control onto your form, an annoying dialog would pop up saying "Type in a meaningful name for this control." But Visual Studio is not interested in having you write good code. It wants you to write code fast.

While I'm on the subject of variable names, I should say something good about Visual Studio. Visual Studio 2005 has a totally splendid variable-renaming facility. You know how sometimes you really want to rename a variable to be more in tune with its actual function in the program but you're afraid to because of possible side effects of search-and-replace? Well, this new variable-renaming avoids all that, and it will also tell you if you're renaming something to an existing name. I hope people take advantage of this to rename their controls to something other than the Visual Studio defaults.

Overused Fields

Another problem with Visual Studio's generated code is that every control is made a field of the class in which it is created. This is a hideous programming practice, and it really bothers me that programmers may be looking at the code generated by Visual Studio to learn proper programming technique, and this is what they see.

In the pre-object oriented days of C programming, we had local variables and global variables, variables inside of functions, and variables outside of functions. One objective of good programming in C was in keeping the number of global variables to a minimum. Because any global variable could be modified by any function, it was easier to get the sense of a program if there simply weren't very many of them — if functions communicated among themselves strictly through arguments and return values.

Sometimes, however, C programmers got a little, let's say, *sleepy*, and made a lot of variables global that didn't really need to be, because it was just easier that way. Rather than add an extra argument to a function, why not just store it as a global?

Global variables are basically gone in object-oriented programming, except that fields are now the *new* global variables, and they can be abused just as badly.

One basic principle of object-oriented programming is hiding data. This is generally meant to apply

between classes. Classes shouldn't expose everything they have. They should have as small a public interface as possible, and other classes should know only what they need to know. But the principle of data hiding is just as important *inside* a class. Methods should have limited access to the data that other methods are using. In general, variables should be made local to a method unless there's a specific reason why they need to be accessed from some other method.

When you create an object using the *new* operator, the amount of storage allocated from the heap for the object must be sufficient to accommodate all the fields defined in the class you're creating the object from and all the ancestor classes. The fields basically define the size of the object in memory. I know that we're long past the point of worrying about every byte of storage, but when you look at the fields you've defined in your class, you should be asking yourself: Does all this stuff really need to be stored with each object in the heap? Or have I successfully restricted the fields to that information necessary to maintain the object?

Someone told me that he likes to store objects as fields because he's afraid the .NET garbage collector will snag everything that's not nailed down into a proper field definition. After years of defining very few fields in .NET, I can tell you this is not a problem. The .NET garbage collector will only delete objects that are no longer referenced anywhere in the program. If you can still access an object in some way, it is not eligible for garbage collection.

In theory, no child control created as part of a form needs to be stored as a field because the parent — generally a form — stores all its child controls in the collection named *Controls*, and you can reference each control by indexing the *Controls* property using and integer or the text *Name* property you assigned to it when creating the control. And here again you're in much better shape if your controls have meaningful names and not *button1*, *button2*, *button3*, *label1*, *label2*, *label3*…

Whether a particular object is defined as a field or a local variable is something that we as programmers should be thinking about with every object we create. A label that has the same text during the entire duration of the form can easily be local. For a label whose text is set from an event handler of some other control, it's probably most convenient to store as a field.

It's as simple as that. But Visual Studio doesn't want you to think about that. Visual Studio wants everything stored as a field.

Visual Studio Demystified

Even if Visual Studio generated immaculate code, there would still be a problem. As Visual Studio is generating code, it is also erecting walls between that code and the programmer. Visual Studio is implying that this is the only way you can write a modern Windows or web program because there are certain aspects of modern programming that only it knows about. And Visual Studio adds to this impression by including boilerplate code that contains stuff that has never really been adequately discussed in the tutorials or documentation that Microsoft provides.

It becomes imperative to me, as a teacher of Windows Forms programming and Avalon programming, to deliberately go in the opposite direction. I feel I need to demystify what Visual Studio is doing and demonstrate how you can develop these applications by writing your own code, and even, if you want, compiling this code on the command line totally outside of Visual Studio.

In my Windows Forms books, I tell the reader not to choose Windows Application when starting a new Windows Forms project, but to choose the Empty Project option instead. The Empty Project doesn't create anything except a project file. All references and all code has to be explicitly added.

Am I performing a service by showing programmers how to write code in a way that is diametrically opposed to the features built into the tool that they're using? I don't know. Maybe this is wrong, but I can't see any alternative.

Dynamic Layout

I've been talking about the release of Visual Studio 2005 next month. Along with Visual Studio is also the .NET Framework 2.0, with some significant enhancements to Windows Forms, including a very strong commitment to "dynamic layout" (sometimes also known as "automatic layout"). Dynamic layout is also a significant part of the design philosophy of Avalon.

Dynamic layout is an example how Web standards have now influenced standalone client Windows programming. We have learned from HTML that it's not necessary to put every single object on the screen with precise pixel coordinates. We have learned that it's possible for a layout manager built into our browsers to flow text and graphics down the page, or to organize content in a table or frames. HTML has also provided a good case study in illustrating some ways *not* to implement these things, so these ways can be avoided.

Basically, when using dynamic layout in Windows Forms, the actual location and size of controls isn't decided until run time. When the form is displayed, the layout manager takes into account the size of the controls, the size of the container, and lays out the controls appropriately.

Dynamic layout under Windows has come to be seen as necessary for a couple reasons. First, we are getting closer to a time when 200 dpi and 300 dpi displays will become available. It would be very nice if Windows programs written today didn't crumble into incoherence on these displays.

Also, as controls have become more complex, they have become more difficult to use in predefined layouts. How big should a particular control be? Perhaps that can only be known at run time. It's much better if the program does not impose a size on the control, but rather that the control determines its own size at run time, and to have that size taken into account during layout.

Now obviously this is more complex, because a layout manager has to interrogate the controls and determine how large they want to be, and then attempt to accommodate them all within an area of screen real estate.

But this is built into the classes of Windows Forms 2.0. The *FlowLayoutPanel* and the *TableLayoutPanel* together with the *SplitContainer* and the *Dock* property provide a full array of tools for dynamic layout. You can basically design entire forms and dialog boxes largely without using pixel coordinates or sizes. There are some exceptions, to be sure. You'll probably want to set widths of combo boxes to specific sizes based on average character widths. But for the most part, this stuff works. In my new book I have a whole chapter on dynamic layout that concludes with code that duplicates the standard *FontDialog* with a *TableLayoutPanel*, and I come pretty close.

We now have the proper tools in the form of classes to render the Visual Studio forms designer obsolete. Unfortunately, nobody has told Visual Studio about this new age. Visual Studio definitely supports layout based on *FlowLayoutPanel* and *TableLayoutPanel*, of course, but you have to know about these things and explicitly put them on your form before you start populating the form with controls. Inexplicably, Visual Studio still stores pixel coordinates and sizes in your source code even though they are irrelevant in these cases

Avalon and XAML

Some people have assumed that with the planned introduction of Avalon next year — now properly called the Windows Presentation Foundation — that Windows Forms is already obsolete. But the more I look at Avalon, the less I believe that to be the case. Certainly Avalon has the power and breadth to support large applications, and a graphics system that will bring 3D capability to the average programmer.

But Avalon is currently missing some amenities we take for granted, such as standard File Open, File Close, and Font dialogs, a split panel, and it has nothing approaching the new and totally astonishing *DataGridView* control in Windows Forms 2.0. Whether these things get into Avalon before it's released, I don't know. But Windows Forms may still be the best bet for quickly constructing forms-based applications for in-house corporate use.

Avalon, as I mentioned, also has a strong commitment to dynamic layout. In Windows Forms, doing layout with pixels locations and sizes is basically the default. You need to specifically create a *FlowLayoutPanel* or *TableLayoutPanel* to dispense with pixels.

In Avalon, working with pixel coordinates is not the default. For each window or page, you have to pick a particular layout model, and the one where you specify pixels — a descendent of *Panel* named *Canvas* — is certainly being deemphasized as a choice. And even if you do go with *Canvas*, you're not really working in units of pixels. You're using something oddly called "device-independent pixels," which is a coordinate system based on units of 1/96th of an inch, which is a one-to-one map with today's default video display settings, but lets us achieve device independence with non-default settings.

But that's not the half of it. When you put together your windows and dialogs in Avalon, you can do it either in code or in XML. Microsoft has come up with an entire page description language called XAML, the Extensible Application Markup Language, pronounced "zammel." In a proper Avalon program, the XAML is supposed to contain all the layout stuff, and the programming code ties it all together, mostly with event handlers.

I was at first skeptical of this invention called XAML. I had liked the Windows Forms approach of doing everything in code, and now something like a resource script — but obviously much more sophisticated — was being reintroduced into Windows programming. I gradually became persuaded, however, and now I like it.

One thing that the Avalon architects are trying to do is let programmers and designers work around common files. Not every software project requires people who design the screens and dialog boxes, but large ones do, and in the past the designers did mockups with graphical tools and used bitmaps to show the programmers what they wanted. With Avalon they can use design tools that create and edit XAML, and the programmers can use this same XAML as source code in creating the executables.

Another reason I was persuaded in this: As we get more into dynamic layout, a form becomes a hierarchy of different types of panels and controls. You might start with a *DockPanel* and put menus and toolbars at the top of a window and a status bar at the bottom, and then inside you put a *Grid* panel, and then perhaps within some of the cells of the grid you have a *StackPanel*, and so forth.

This hierarchy of panels and controls has an analogue in the XAML as a nesting of elements. If the XAML is indented in a standard way, you can see the hierarchy right there in the file.

I also have much less of a problem with a visual designer generating XAML than generating C# code. As we've seen, Visual Studio doesn't even want you to look at the C# code it generates, and for good reason. But XML is different. It was a design point of XML that it can be created and edited by either human or machine, and it shouldn't matter at all which is which. As long as the XML is left in a syntactically and semantically correct state — and there is no middle ground — it shouldn't matter who edits it and in what order. If I'm going to be putting XAML in a book, it doesn't matter if I wrote it or Visual Studio wrote it, because it should basically be the same. And, for someone who wants to recreate the layout of panels and controls in Visual Studio, the XAML serves as a clear guide in illustrating the hierarchy.

Will there be a designer built into Visual Studio that works entirely with XAML and lets me write my own C# code? I don't know yet. It certainly makes sense.

I am also hoping that developers eventually achieve a good balance between XAML and code. The Avalon architects are so proud of XAML — and rightfully so — that they tend to use it for everything. I saw an Avalon clock application that somebody at Microsoft wrote. It actually set the time once in code and used Avalon animation entirely implemented in XAML to keep the clock going. It was very, very cool, except that the 12 tick marks of the clock were implemented in 12 virtually identical chunks of XAML. The only thing that would have appalled me more was seeing 60 tick marks implemented 60 identical chunks of XAML.

I don't know what rule you go by, but for me it's always been simple: "Three or more: Use a *for*." This is why we have loops. This is why we are programmers.

The Pure Pleasures of Pure Coding

A couple months ago — perhaps as an antidote to all this highfalutin Windows Forms and Avalon programming I've been doing — I started coding in C again. Just a little bit. A magazine I read — the British weekly *New Scientist* — runs a tiny regular feature called "Enigma," which presents a little math puzzle. Here's a particularly short one from this past June: "What is the largest integer whose digits are all different (and do not include 0) that is divisible by each of its individual digits?"[8] If you solve one of the Enigma problems, you can send in your solution, and one is chosen at random on a particular date, and if yours is picked you get 15 pounds and a mention in the magazine.

These Enigma problems have actually annoyed me for years because they always seemed solvable by writing a short program and testing out the possibilities, and for some reason that seemed absurd to me.

A few months ago I decided it might actually be interesting solving the problems with code, and then posting the programs and solutions on my web site the day after the deadline, but about a week before the answer appears in print.

I decided to use plain old ANSI C, and to edit the source code in Notepad — which has no IntelliSense and no sense of any other kind — and to compile on the command line using both the Microsoft C compiler and the Gnu C compiler.

What's appealing about this project is that I don't have to look anything up. I've been coding in C for 20 years. It was my favorite language before C# came along. This stuff is just pure algorithmic coding with simple text output. It's all content.

I also discovered is that the problems *do* require some thought before you code them up. Very often, the total number of combinations is prohibitive. You really have to whittle them down. You don't want to write a program that runs for a week before getting to the *printf* statement. For example, take the problem I quoted: "What is the largest integer whose digits are all different (and do not include 0) that is divisible by each of its individual digits?" It really helps to realize that this integer cannot contain the digit 5 if it also contains any even digit, because then the number would be divisible by 10, and the last digit would be 0, which isn't allowed. So, the number we're looking for probably doesn't include 5. That immediately reduces the possibilities by an order of magnitude.

Even after this preliminary process, there's still coding to do, but there's no APIs, there's no classes, there's no properties, there's no forms, there's no controls, there's no event handlers, and there's definitely no Visual Studio.

It's just me and the code, and for awhile, I feel like a real programmer again.

---

[1]It was the late 70s. I was working for New York Life Insurance Company at 51 Madison Avenue, programming in PL/I on an IBM 370 via TSO (Time Sharing Option) and an IBM 3278 terminal. I was using a new-fangled "full-screen editor" called FSE to edit my source code. At one point I pressed the Backspace key and instead of the cursor moving backwards as is normal, the entire screen moved left one character, including the frame of the editor, with the beginning of each line wrapping to the end of the preceding line. It was as if the entire frame buffer shifted. I pressed the Backspace key a couple more times, and it did the same thing. Then it started going by itself, with the lines circling around the screen and right up through the top until the screen was entirely clear. Then we called the IBM repair guy. I was very eager to tell him what happened and hear his diagnosis, but that's not how they repaired 3278s. Instead, the repair guy had a box full of spare circuit boards. He swapped them in and out of the machine until the terminal worked again.

[2]Others can be found on the page "Hollywood & Computer" from the Charles Babbage Institute web site: http://www.cbi.umn.edu/resources/hollywood.html.

[3]"Taglines for Colossus: The Forbin Project," http://www.imdb.com/title/tt0064177/taglines.

[4]"Focus: Misguided Missiles," *The Onion* (New York City print edition), vol. 41, no. 30, pg. 15.

[5]Manually counted in "Quick Reference" section of *Microsoft Windows Software Development Kit: Programmer's Reference* (Microsoft Corporation, 1985).

[6]Charles Petzold, *Programming Windows 95* (Microsoft Press, 1996), pg. 17.

[7]A package of 500 Staples brand cards is approximately 4 inches.

[8]"Enigma 1343: Digital Dividend," *New Scientist*, 4 June 2005, 28.

First Posted: October 2005

Back to the Et cetera page.

Back to Home