

# The SKILLFUL DEVELOPER

By Mike Gunderloy

Real-world advice  
for taking solution  
development to  
the next level.

# The Skillful Developer

---

*Real-world advice for taking solution development to the next level.*

**By Mike Gunderloy**

*Author of the Developer Central Newsletter  
<http://lists.101com.com/nl/main.asp?NL=adt>*

**Published by CertCities.com**  
**a division of 101communications LLC**  
16261 Laguna Canyon Rd  
Irvine, Calif. 92618  
<http://CertCities.com>, [feedback@certcities.com](mailto:feedback@certcities.com)

**Publisher:** Henry Allain  
**Editorial Director:** Dian Schaffhauser  
**Editor:** Becky Nagel  
**Marketing Manager:** Michele Imgrund  
**Senior Web Developer:** Rita Zurcher

ISBN#: 0-9743828-0-9

*Copyright (c) 2003 101communications LLC. All rights reserved. Except as permitted by law, this publication may not be reproduced or redistributed in any means in part or whole without the express written permission of 101communications.*

## Table of Contents

Introduction.....	3
Plea for Mindful Development .....	5
What Was Still Is .....	10
Filling Your Toolbox.....	15
Pragmatic Programming—Is It For You? .....	20
Improve Your Environment.....	25
Is Certification the Answer? .....	30
Software Developer’s Resource Guide.....	35
Extreme Programming Evaluated .....	40
Your Testing Environment.....	45
The Shape of Things .....	50
A Look at .NET.....	55
<i>PeopleWare</i> : Your Father’s Software Development?.....	60
The State of Software Development .....	65
Rapid Development.....	70
Refilling Your Toolbox.....	75
The Code Handyman .....	80
Language Wars .....	85
Baking in Security.....	89
Web Services: Worth Your While? .....	93
Give the Users What They Want .....	99
Home for the Holidays .....	104
Are You a Hacker?.....	109
The Essential Infrastructure .....	114
Two Heads are Better Than One?.....	119
The State of Software Development .....	124
Beyond the Code.....	128
Back to the Toolbox.....	132
Begin at the Beginning.....	138
Something New Under the Sun.....	142
Taste-Tested Coding .....	147

## Introduction

The title of this book might need a bit of explanation. Yes, this collection of my columns from MCPmag.com is about becoming a more skillful developer—but perhaps not in the way that you normally think about skill. You won't find any new algorithms (or even old ones) here, or tutorials in writing properly object-oriented code. You won't find guides to new languages or advice on which class library to use in a particular situation. What you will find is book reviews, tool recommendations, and some thinking about what separates successful developers from the rest of the pack.

You see, I draw a distinction between *coders* and *developers*. If you've just been through a college course and learned the ins and outs of C++, and passed the final exams and learned to write code to solve any given problem, you're a darned good coder. You understand the syntax of the language and how to use it. But when you get out into the real world of actually turning code into applications that people will use and pay you for, you'll discover that just banging out code is not enough. Learning the rest of the job is what separates developers from coders.

The rest of the job includes things like planning your work, writing unit tests, writing documentation, fitting in with a team, choosing the right tools for the job, source code control, bug tracking, automated builds, and a host of other things. Not every developer will be an expert in all of these fields, of course. But over time, I've come to some conclusions about the

## The Skillful Developer

superior developers. In large part, the great developers are the ones who treat software development as a craft that you can never stop learning. They're the ones who are always ready to try out a new tool or technique, investigate a new language, or discuss development with their peers.

Some of skill in software development is simply a matter of experience, of course. The more source code you've written or reviewed, the more you have to draw on the next time you run up against an apparently -new problem. But I firmly believe that development skill can also be learned. If you take seriously the goal of becoming familiar with the field and trying out new ideas, you will end up with skill out of proportion to your experience. I hope you'll find at least a few pointers in this collection to things that intrigue you, or that provoke you enough to learn more in your quest for skill.

These essays appear just as they did on the MCP Magazine Web site, though I've taken the liberty of fixing a problem where some columns were posted out of order, and corrected a few URLs. Otherwise, I've resisted the temptation to do any wholesale revising. Instead, you'll find my current reflections at the end of each essay. If these, or anything else in the collection, inclines you to comment, I'd love to hear from you. My e-mail address is [MikeG1@larkfarm.com](mailto:MikeG1@larkfarm.com).

*Mike Gunderloy  
Endicott, Washington  
October 2003*

## Plea for Mindful Development

*Zen has the concept of mindfulness, embodied in part by paying attention to each breath. Good software development means paying attention to every single line of code.*

Recently I was writing some code to retrieve information from SQL Server 2000 Analysis Services via the query retrieval objects in the ADO/MD library. Things were going along fine, until suddenly Visual Basic told me “Error 13: Type mismatch.”

“Hmmm,” I thought, “I don’t see how there can be a type mismatch in this code.” But then I remembered a problem with using Recordset variables in Visual Basic, and converted things to use late-bound Object variables instead. The problem continued, and I coded up a routine to dump the ADO Errors collection. I stuck that routine at the end of my error trap, and it never got called. Well, that’s bad. Could I be doing something so obscure with ADO/MD that I was blowing up some internal variable in Visual Basic?

The answer to that question proved to be “no.” After an hour of messing around, I found out that I wasn’t doing anything sophisticated and obscure. I was doing something obvious and stupid. Here’s the error-handling routine that was built into my Visual Basic code:

```
ExitHere:
    Exit Sub
HandleErr:
    MsgBox "Error " & Err.Number & ": " & _
        Err.Description, "cmdBasicInformation"
    Resume ExitHere
    Resume
End Sub
```

If you're not a Visual Basic programmer, it may not be blindingly obvious what's wrong with this code. Here's what it should have been:

```
ExitHere:
    Exit Sub
HandleErr:
    MsgBox "Error " & Err.Number & ": " & _
        Err.Description, , "cmdBasicInformation"
    Resume ExitHere
    Resume
End Sub
```

Yes, a single comma had cost me an hour's work and made me feel very stupid to boot. The bug wasn't in the code I was executing at all: It was in my error-handling routine, which was trying to send a string constant to an integer parameter. After 25 years of writing computer programs, I had inserted a bug into my code in one of the easiest ways to do so: by not paying attention to the code that I was writing. I've probably written 10,000 error-handling routines in Visual Basic by now. No doubt that lulled me into the false sense of confidence: "I'll never make a mistake in this code, so I don't need to test it." (Actually, the problem was even worse than that. I'd made the mistake the day before in another procedure, and had since copied the mistake into dozens of places in my program. But that's a lesson for another day.)

So, welcome to my new column for *MCP Magazine* Online. It takes a certain amount of chutzpah to start a series on improving software development by confessing to a boneheaded mistake. But you know what? Boneheaded mistakes are the norm in our field of writing software. Every developer I've ever worked with (and I've worked with some good developers indeed) has written code with bugs in it. What distinguishes the good developers is that they find the bugs and fix them. Sometimes they even do so quickly.

Overall, I'd say, the state of software development is not good. Let's look at Microsoft, for example. There are nearly 300,000 articles in the Microsoft KnowledgeBase these days. At least one third of these are reports of bugs—some with fixes, some with workarounds, and some that you just have to live with. That's 100,000 bugs. That's 4,000 bugs in released software for every year that Microsoft has been in business. That's a lot of bugs. And that's only the ones that they confess to publicly. I'd guess there are at least 10 times that many bugs in the internal bug-tracking databases at Microsoft.

Microsoft hires some of the best software developers around. I've worked with some of them, and their skills are amazing. These are people who can hold a myriad of details in their minds at once, who can write code in their sleep, who are capable of blinding flashes of insight that result in brilliant new algorithms. They're supported by a company whose entire philosophy revolves around insulating developers from nonsense so that they can concentrate on writing code. And yet they still crank out code with bugs, year after year after year. If that's the best that Microsoft can do, is there hope for you and me?

In a word, yes. Software development has been around for more than 50 years now, and in that time we've learned a lot about what works and what doesn't work. There are resources out there, books and tools that will help you become a better developer (that is, one who releases code with fewer bugs in it). In this column, I'm going to try to introduce you to some of the resources, practices, and tools that have helped me over the years.

Let me start by making a somewhat arbitrary distinction: There are code mechanics, and there are mindful developers. My goal is to make you a more mindful developer. What does that mean? Step out of the software field for a moment to consider this quote from Buddhist monk Thich Nhat Hanh:

*I remember a short conversation between the Buddha and a philosopher of his time.*

*"I have heard that Buddhism is a doctrine of enlightenment. What is your method? What do you practice every day?"*

*"We walk, we eat, we wash ourselves, we sit down."*

*“What is so special about that? Everyone walks, eats, washes, sits down ...”*

*“Sir, when we walk, we are aware that we are walking; when we eat, we are aware that we are eating ... When others walk, eat, wash, or sit down, they are generally not aware of what they are doing.”*

*In Buddhism, mindfulness is the key. Mindfulness is the energy that sheds light on all things and all activities, producing the power of concentration, bringing forth deep insight and awakening. Mindfulness is at the base of all Buddhist practice.*

No, I'm not suggesting that you need to become a Buddhist to write better code (though I wouldn't be at all surprised if it helped). What I am suggesting is that you need to become more mindful of the code that you write. A code mechanic knows all about the language he's working with and can crank out endless lines of code effortlessly, without thinking about it. And that's the problem! When the mindful developer is writing code, she is aware that she is writing code. She's thinking about what the code should do. She's writing each error handler for the first time, not writing the same error handler for the ten-thousandth time.

There may be a perfectly mindful developer out there somewhere who writes code correctly the first time and every time. For the rest of us, though, there's more to the process of software development. Look back at my original bug at the start of this column. Two things came together to waste an hour of my time:

1. I wrote a bad line of code.
2. I didn't test the line of code that I wrote.

The second point is subtle. Yes, in one sense I tested the bad line of code: I executed it, and Visual Basic obligingly returned an error. But my "testing" was at the crudest of levels. I ran the code, and something broke. Then I changed the code, ran it again, and something broke. Lather, rinse, repeat.

This is classic "black box" testing, and it's the way that Quality Assurance departments around the world test code. Throw some inputs at the code and see if it breaks. If it breaks, send it back to the developer with

a bug report. But as developers, we have a better way to test. We can do white box testing.

The goal of white box testing is simple: It's to exercise every line of source code. We can do this because we have the source code and we have a powerful tool for exercising it. That's the debugger. Any modern computer language has some way to step through lines of code one at a time, watching their actions in human time instead of in computer time. Steve Maguire points out the power of this technique in his book, *Writing Solid Code* (Microsoft Press, 1993):

*The best way to catch bugs is to look for them the moment you write or change code. And what's the best way programmers can test their code? It's by stepping through it and taking a microscopic look at the intermediate results. I don't know many programmers who consistently write bug-free code, but the few I do know habitually step through all of their code.*

Sounds easy, doesn't it? But like any other bit of mindfulness, single-stepping through all of your code takes discipline. It's easy to get into a mindset of "Oh, that code is obvious. I don't need to test it." Beware! There is only a single comma between this mindset and an hour of wasted work. I know it will seem like you're wasting time when you first start this practice. But stick with it, and you'll save time in the long run—and you'll know your own source code better, and increase your confidence that it works correctly. And surely that is worth a little extra effort.

*Originally published in April 2001.*

**Reflections:** There's another lesson here, although I didn't tease it out in the original column: Why on earth was I writing yet another error-handling routine by hand? That sort of repetitive busywork is what good development tools should protect us from. These days, I'd look to a code-generation tool to allow me to write the error-handling code once and then insert it whenever it's needed.

Also, I've come to believe that test-driven development (writing the unit tests before writing the code) is at least as important as single-stepping through code if you want to write bug-free code. See the essay "Taste-Tested Coding" on page 148 of this guide for some thoughts about test-driven development.

## Improve Your Environment

*Don't overlook the role your physical space plays in the software that you create.*

Almost 15 years ago, Tom DeMarco and Timothy Lister studied programmer productivity in a variety of environments. They came to the conclusion, based on their own work plus data from such places as IBM and Bell Labs, that developers produced more and better code in offices than in cubicles. They suggest that this is directly related to the ease (or difficulty!) of getting into a “flow” state where you get deeply involved with the task at hand. In their book *Peopleware: Productivity and Teams*, DeMarco and Lister address employers:

*In most of the office space we encounter today, there is enough noise and interruption to make serious thinking virtually impossible. More is the shame: Your people bring their brains with them every morning. They could put them to work for you at no additional cost if only there were a small measure of peace and quiet in the workplace.*

DeMarco and Lister recommend private offices with doors for developers, a bit of wisdom that has been endorsed by many writers since even as cubicles have become ever more the norm.

### **Reviewed**

*Peopleware: Productivity and Teams*  
By Tom Demarco and  
Timothy Lister  
Dorset House  
\$33.95, ISBN 0-93263-343-9

*Extreme Programming Explained*  
By Kent Beck  
Addison-Wesley  
\$28.95, ISBN 0-20161-641-6

On the other hand, Kent Beck takes a different view in *Extreme Programming Explained*. His style of software development involves lots of teamwork, and he argues that you need a workspace that facilitates teamwork. He likes bullpens with large tables for the computers, little cubbies for personal space, and a communal relaxation space with toys and a coffeemaker. And of course he can back his ideas up with data about how much more productive it makes developers as well.

### **Which Approach Is Better?**

So, have things really changed that much in 15 years? Nope! DeMarco & Lister and Beck come to different conclusions about workable space because they have different models of software development in mind. If you're writing a constrained piece in a large architecture and need to concentrate to get a tricky algorithm perfect, you want an office with a door that closes and a lack of hubbub. If you're using pair programming and other XP techniques to blast through rapid iterations of code, then getting a good team together where they can develop some synergy is essential. But all three authors have the same attitude towards facilities. Here's a bit from Beck's book:

*If the corporate attitude towards facilities is at odds with the team's attitude, the team wins. If the computers are in the wrong place, they are moved. If the partitions are in the way, they are taken down. If the lights are too bright, they are taken out. If the phones are too loud, one day, mysteriously, they are all found to have cotton stuffed in the bells.... All this screwing around with furniture can get you in trouble.... I say, "Too bad." I have software to write, and if getting rid of a partition helps me write that software better, I'm going to do it. If the organization can't stand that much initiative, then I don't want to work there, anyway.*

So why, 15 years after we learned that facilities design makes a difference to software productivity and quality, do so many of us still work in frankly inadequate spaces? There are, I think, two reasons for this. First, it's easier to see

the cost of good facilities than it is to see the cost of bad facilities. Second, we get our sense of pecking order mixed up with everything we do.

### **Rats in a Maze**

At one point, I took a consulting contract to help a small team within a large organization with a Y2K remediation project. Their workspace was drearily typical of corporate America: acres of identical cubicles. Well, not quite identical. For reasons known only to themselves, the facilities people moved partitions around from time to time, creating little workgroup islands or rearranging corridors. As a new visitor, I needed a native guide to find anything for the first week or two. Even after that, it was perfectly possible to turn a corner on a route you were used to and find a blank wall in front of you.

But it gets better than that. There was little attempt to keep groups together. The corporation had barely enough cubicles for all the workers, so when someone new was hired, they were slotted into the first available cubicle. That meant, in most cases, they were nowhere near their “team.” In the case of the folks I was working with, we were spread out over several buildings, with a long walk for some people to the shared conference room. You can imagine how rarely we saw each other in person, and how hard it was to stay in touch with the whole project. Not surprisingly, coordination was difficult or impossible, and it was not unusual for work to fall in the cracks or be duplicated.

Now, I’m sure the facilities folks could have shown us numbers on how much money this rat-in-a-maze approach saved, compared to actual offices or even bullpen settings. And I’m sure it was much cheaper to scatter folks willy-nilly, instead of adding some extra space and keeping groups together. But I’d guess that every dollar saved on space in that corporation wasted at least ten dollars in developer salaries. Little wonder that the project did not go well.

### **To the Managers Go the Spoils**

As an example of my second factor, I think back to one of the first jobs I had, with a large computer manufacturer (since gone bankrupt) at the very start of the PC revolution. The company had been around long enough that everything from the chair you got, to whether you got a cubicle or an office (and whether the office had a door or a window, how many chairs you got for visitors, and so on) was rigidly set in the company hierarchy. Get a promotion, get a bigger office. Get another promotion, you actually got a PC on your desk. Yes, the computers went to the managers first.

The group I happened to be working for at the time was responsible for programming the machines that stuffed integrated circuits into circuit boards to make the PCs. We used punched paper tape (I know, I'm dating myself) to move programs from our office area to the shop floor. After some research, we decided that we could improve production by putting in a PC and using a rudimentary network to program the shop floor machines, disposing with all that abysmal paper tape.

Well, you guessed it. We weren't upper-level managers, so we couldn't get a PC. It didn't matter that it would improve productivity; the prestige rules said we couldn't have one. Fortunately for that company, we found an innovative solution: one weekend we swiped the guts out of the division manager's PC, found a spare monitor and a spare power supply, and breadboarded our own production machine. Since the division manager never actually turned on his PC, this worked out to be an ideal solution.

So, what's the lesson here? I think there are several things to learn from what we know about space design for developers:

1. Space matters. There's no one-size-fits-all solution; the bottom line is that developers produce better software when their workspace is appropriate for their methodology, whether that means private offices or bullpens. But five-foot-high anonymous cubicles do not seem to be good for any development method.
2. In many cases, we're stuck with bad space because we haven't been able to make an argument for good space. If there are concrete improvements that can be made to your working space—perhaps elimination of an extra wall, a more comfortable chair, or a less-obnoxious phone—it's worth documenting, as best you can, the effect of the poor facilities on your work. If you keep track of time wasted running through the maze to find a co-worker, time lost getting back into the flow after the obnoxious phone rings, and so on, you might be able to convince a reasonable boss to fight with the corporate facilities people on your behalf.
3. Of course, this can be a dangerous tactic in these downsizing times—sometimes the squeaky wheel gets removed from the axle entirely!