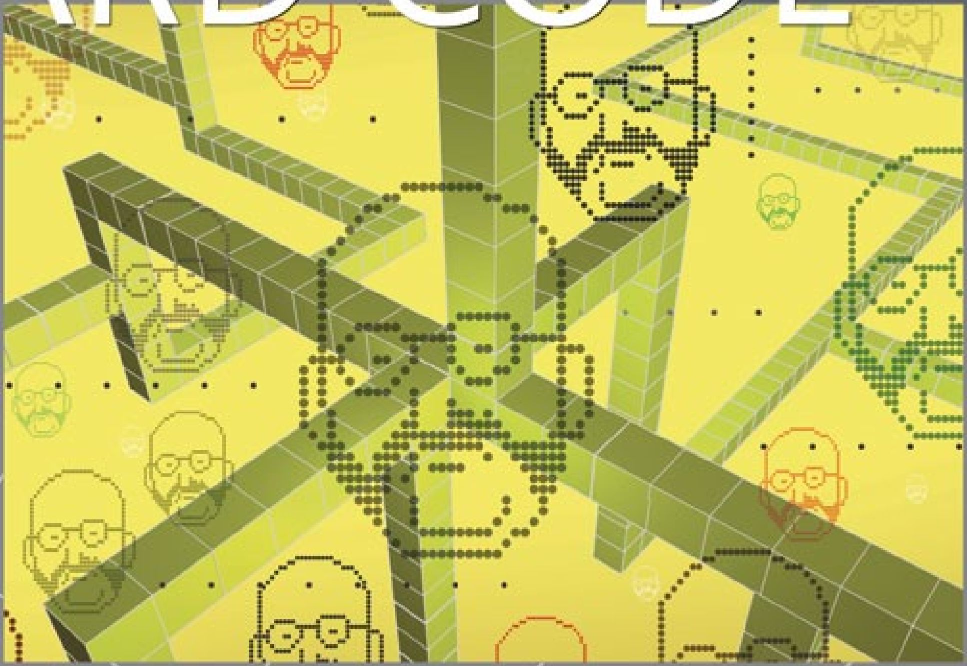
Microsoft

BEST PRACTICES

2
SECOND EDITION

I. M. WRIGHT'S "HARD CODE"



A Decade of Hard-Won Lessons from Microsoft®

Eric Brechner

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2011931649
ISBN: 978-0-7356-6170-7

Printed and bound in the United States of America.

First Printing

Acquisitions and Developmental Editor: Devon Musgrave
Project Editor: Valerie Woolley
Editorial Production: Curtis Philips
Copyeditor: John Pierce
Indexer: William Meyers
Cover: John Hersey

Reader Acclaim for I. M. Wright's "Hard Code" Column

Any large organization is prone to fall prey to its own self-made culture. Myths about how things should be or should be done turn into self-fulfilling prophecies. Any such trend is surely terminal for any organization, but it is a rapid killer in a technology company that thrives on perpetual innovation. Eric Brechner does an incredible job at pulling out the scalpel and cutting deep into such organizational fluff. He is also not shy at throwing a full punch—the occasional black eye being an intended outcome. While some of the lingo and examples are somewhat more appealing to the Microsoft insider, there is little in his wit and wisdom that shouldn't become lore across the software industry.

—Clemens Szyperski, Principal Architect

Great article on dev schedules by "I. M. Wright." It applies equally well to infrastructure projects that my group is involved in.

—Ian Puttergill, Group Manager

You're not getting any death threats or anything, are you?

—Tracey Meltzer, Senior Test Lead

This has got to be a joke—quite frankly, this type of pure absurdity is dangerous.

—Chad Dellinger, Enterprise Architect

Eric is a personal hero of mine—largely because he's been the voice of reason in the Dev community for a very long time.

—Chad Dellinger, Enterprise Architect

Software engineers can easily get lost in their code or, even worse, in their processes. That's when Eric's practical advice in "Hard Code" is really needed!

—David Greenspoon, General Manager

I just read this month's column.... I have to say this is the first time I think you are pushing an idea that is completely wrong and disastrous for the company.

—David Greenspoon, General Manager

You kick ass Eric :) I was having just this conversation with my PUM and some dev leads a few months ago. Great thinking piece.

—Scott Cottrille, Principal Development Manager

We really like these columns. They are so practical and, well, sane! I also love that I can refer back to them when I'm trying to help a junior dev get up to speed and they remember the column since they are usually so entertaining.

—Malia Ansberry, Senior Software Engineer

Nice job, Eric. I think you really hit the nail on the head in this column. I think a good message to give to managers is, "Don't be afraid to experiment." How things really work is so different than idealized theories.

—Bob Fries, Partner Development Manager

I just wanted to let you know how much I love what you write—its intelligent, insightful, and you somehow manage to make serious matter funny (in the good way).

—Niels Hilmar Madsen, Developer Evangelist

We're going to be doing the feature cuts meetings over the next few weeks, and your death march column was just in time. It's a good reminder of lessons that were hard learned but somehow are still more easily forgotten than they should be.

—Bruce Morgan, Principal Development Manager

I wanted to let you know that I really appreciated and enjoyed all of your writings posted on the EE site. Until, today, I read ["Stop Writing Specs"]. I have to say that I strongly disagree with your opinion.

—Cheng Wei, Program Manager

Who are you and what have you done with Eric Brechner?

—Olof Hellman, Software Engineer

Eric, I just read the "Beyond Comparison" article you wrote and want you to know how much I appreciate that you actually communicated this to thousands of people at this company.... Thank you for your passion in managing and leading teams the right way and then sharing the HOW part of that!

—Teresa Horgan, Business Program Manager

Contents at a Glance

Table of Contents

Foreword

If you want to know about etiquette, you turn to Miss Manners. If you're having trouble with your love life, you might turn to Dear Abby. If you want to know what's going on at Microsoft and how one bullhead named I. M. Wright approaches things, then this is a book for you. I. M. Wright is also known around Microsoft as Eric Brechner.

Building software is a challenge. I've always considered it a creative team sport, one that requires you to remember not only what worked in the past but what didn't. When I worked at Microsoft, Eric was my sounding board. When I got stuck or frustrated, he seemed to know just what to say to help me. And sometimes I didn't even have to ask. Just when I had reached the point where I knew I needed help, an I. M. Wright column would pop up that addressed my concerns—issues that were common at Microsoft and at any software development organization.

Having a copy of *Hard Code* is like having Eric just around the corner from your office. Having trouble dealing with change? Eric has an answer. Does your team have low morale? I bet Eric has a bit of advice for you. Quality issues plaguing your code? I know Eric can help. Writing as I. M. Wright, Eric tackles the tough problems of software development in a lighthearted way that makes you smile while making you think.

And Eric doesn't write just about the problems you're likely to encounter. He also highlights lessons learned from the many successful practices he's seen as part of a company that builds and releases products and services used by millions of people around the globe. This new edition includes a gold mine of new advice and success stories—valuable nuggets that I often make required reading for my customers. *Hard Code* is a rare gem, one that everyone should have on his shelf.

Mitch Lacey
Consultant and former Microsoft employee
Mitch Lacey & Associates, Inc.
May 2011

Foreword to the First Edition

I was a regular reader of Eric Brechner's columns, penned under the name I. M. Wright, when I met him for the first time. It took me a moment to be sure that I was talking to the same person, since Mr. Wright was notably opinionated, and the modest, polite, and friendly person I was talking to seemed more like Clark Kent.

My favorite columns focused on the relationship between the technical and interpersonal dynamics of people building software in teams at Microsoft. I'm often surprised, given the amount of material that has been written about the company, how much of the story remains untold.

Software engineering managers on large projects have three fundamental problems. First, program code is much too easy to change. Unlike mechanical or civil engineering, where the cost to make a change to an existing system involves actually wrecking something, software programs are changed by typing on a keyboard. The consequences of making an incorrect structural change to the piers of a bridge or the engines of an airplane are obvious even to nonexperts. Yet experienced software developers argue at length about the risks of making changes to an existing program, and often get it wrong.

Construction metaphors actually work quite well for software. Lines of program code can be characterized along an axis of "foundation, framing, and trim," based on their layer in the system. Foundation code is highly leveraged but difficult to change without a ripple effect. Trim is easier to change and needs to be changed more often. The problem is that after a few years of changes, complex programs tend to resemble houses that have been through a few too many remodels, with outlets behind cabinets and bathroom fans that vent into the kitchen. It's hard to know the unintended side effect or the ultimate cost of any given change.

The second fundamental problem is that the industry is so young that the right standards for reusable software components really haven't been discovered or established. Not only have we not yet agreed that studs should be placed 16 inches apart to accommodate either a horizontal or vertical 4×8-feet sheet of drywall or plywood, we haven't really decided that studs, plywood, and drywall in combination are preferable to some yet to be invented combination of mud, straw, rocks, steel, and carbon fiber.

The final problem is really a variation of the second problem. The software components that must be reinvented on every project must also be named. It's customary in the software industry to invent new names for existing concepts and to reuse existing names in new ways. The unspoken secret in the industry is that a nontrivial number of discussions about the best way to build software actually consist of groups of people who use different names and haven't the foggiest idea what each other is saying.

On the surface, these are easy problems. Create some standards and enforce them. In the fast-paced world of high-volume, high-value, low-cost software, this is a great way to go out of business. The reality is that software's greatest engineering liability is also its greatest strength. Ubiquitous software, running on low-cost personal computers and the Internet, has enabled innovation at a breathtaking pace.

As Microsoft grew, the company didn't always have the luxury of researching the best engineering practices and thoughtfully selecting the best qualities of each. The success of the personal computer and Windows transformed the company from working on small projects in traditional ways to writing the book on the largest, most complex software ever developed.

Microsoft faces a continuous struggle to create the optimal system that balances risk against efficiency and creativity. Given the enormous complexity of some of our projects, these efforts can be amazingly heroic. Over time, we've created specialists and organizations of specialists, all devoted to the single hardest problem in the industry, "shipping." We have acquired folklore, customs, cultures, tools, processes, and rules of thumb that allow us to build and ship the most complex software in the world. Being in the middle of this on a day-to-day basis can be thrilling and frustrating at the same time. Eric's columns are a great way to share and learn with us.

Mike Zintel
Director of Development
Windows Live Core
Microsoft Corporation
August 2007

Introduction

For Bill Bowlus, who said, "Why don't you write it?"

For my wife, who said, "Sure, I'll edit it."

You've picked up a best practices book. It's going to be dull. It might be interesting, informative, and perhaps even influential, but definitely dry and dull, right? Why?

Best practice books are dull because the "best" practice to use depends on the project, the people involved, their goals, and their preferences. Choosing one as "best" is a matter of opinion. The author must present the practices as choices, analyzing which to use when for what reasons. While this approach is realistic and responsible, it's boring and unsatisfying. Case studies that remove ambiguity can spice up the text, but the author must still leave choices to the reader or else seem arrogant, dogmatic, and inflexible.

Yet folks love to watch roundtable discussions with arrogant, dogmatic, and inflexible pundits. People love to read the pundits' opinion pieces and discuss them with friends and coworkers. Why not debate best practices as an opinion column? All you need is someone willing to expose themselves as a close-minded fool.

How This Book Happened

In April of 2001, after 16 years of working as a professional programmer at places such as Bank Leumi, Jet Propulsion Laboratory, GRAFTEK, Silicon Graphics, and Boeing, and after 6 years as a programmer and manager at Microsoft, I transferred to an internal Microsoft team tasked with spreading best practices across the company. One of the group's projects was a monthly webzine called *Interface*. It was interesting and informative, but also dry and dull. I proposed adding an opinion column.

My boss, Bill Bowlus, suggested I write it. I refused. As a middle child, I worked hard at being a mediator, seeing many sides to issues. Being a preachy practice pundit would ruin my reputation and effectiveness. Instead, my idea was to convince an established, narrow-minded engineer to write it, perhaps one of the opinionated development managers I had met in my six years at the company.

Bill pointed out that I had the development experience (22 years), dev manager experience (4 years), writing skills, and enough attitude to do it—I just needed to release my inner dogma. Besides, other dev managers had regular jobs and would be unable to commit to a monthly opinion piece. Bill and I came up with the idea of using a pseudonym, and I. M. Wright's "Hard Code" column was born.

Since June of 2001, I have written 91 "Hard Code" opinion columns under the name "I. M. Wright, Microsoft development manager at large" for Microsoft developers and their managers. The tagline for the columns is "Brutally honest, no pulled punches." They are read by thousands of Microsoft engineers and managers each month.

The first 16 columns were published in the *Interface* internal webzine, with many of the topics assigned to me by the editorial staff, Mark Ashley and Liza White. Doctored photos of the author were created by me and Todd Timmcke, an *Interface* artist. When the webzine came to an end, I took a break but missed writing.

I started publishing the columns again 14 months later on internal sites with the help of my group's editing staff: Amy Hamilton (Blair), Dia Reeves, Linda Caputo, Shannon Evans, and Marc Wilson. Last November, I moved all the columns to an internal SharePoint blog.

In the spring of 2007, I was planning to take a sabbatical awarded to me some years before. My manager then, Cedric Coco, gave me permission to work on publishing the "Hard Code" columns as a book during my time off, and Ben Ryan from MS Press got it accepted. The first edition of this book was published later that year.

In addition to the people I've already mentioned, for the first edition I'd like to thank the other members of the *Interface* staff (Susan Fairo, Bruce Fenske, Ann Hoegemeier, John Spilker, and John Swenson), the other people who helped get this book published (Suzanne Sowinska, Alex Blanton, Scott Berkun, Devon Musgrave, and Valerie Woolley), my management chain for supporting the effort (Cedric Coco, Scott Charney, and Jon DeVaan), my current and former team members for reviewing all the columns and suggesting many of the topics (William Adams, Alan Auerbach, Adam Barr, Eric Bush, Scott Cheney, Jennifer Hamilton, Corey Ladas, David Norris, Bernie Thompson, James Waletzky, Don Willits, and Mitch Wyle), and Mike Zintel for being so kind in writing the foreword.

For the second edition, I'd like to highlight the crew of reviewers and long-time readers who keep me from shoving my foot too deeply down my throat each month (Adam Barr, Bill Hanlon, Bulent Elmaci, Clemens Szyperski, Curt Carpenter, David Anson, David Berg, David Norris, Eric Bush, Irada Sadykhova, James Waletzky, J. D. Meier, Jan Nelson, Jennifer Hamilton, Josh Lindquist, Kent Sullivan, Matt Ruhlen, Michael Hunter, Mitchell Wyle, Philip Su, Rahim Sidi, Robert Deupree (Jr.), William Adams, and William Lees); James Waletzky, who wrote two columns for my readers while I was on sabbatical; Adam Barr and Robert Deupree (Jr.), who cajoled me into recording a podcast for my column and helped produce it; Devon Musgrave and Valerie Woolley, who got the second edition published; my managers (Peter Loforte and Curt Steeb) for supporting my efforts; Mitch Lacey for writing the second edition's foreword; and my wife, Karen, who stepped up to edit my columns when I left my editing staff to join Xbox.com.

Finally, I'd like to thank my transcendent high school English teacher (Alan Shapiro) and my readers who are so generous with their feedback. And most of all Karen and my sons, Alex and Peter, for making everything I do possible.

Who Should Read This Book

The 91 opinion columns that make up this book were originally written for Microsoft software developers and their managers, though they were drawn from my 32 years of experience in the software industry with six different companies. The editors and I have clarified language and defined terms that are particular to Microsoft to make the writing accessible to all software engineers and engineering managers.

The opinions I express in these columns are my own and do not represent those of any of my current or previous employers, including Microsoft. The same is true of my asides and commentary on the columns and this introduction.

Organization of This Book

I've grouped the columns by topic into 10 chapters. The first six chapters dissect the software development process, the next three target people issues, and the last chapter critiques how the software business is run. Tools, techniques, and tips for improvement are spread throughout the book, and a glossary and index appear at the end of the book for your reference.

Within each chapter, the columns are ordered by the date they were published internally at Microsoft. The chapters start with a short introduction from me, as me, followed by the columns as originally written by my alter ego, I. M. Wright. Throughout the columns, I've inserted "Eric Asides" to explain Microsoft terms, provide updates, or convey additional context.

The editors and I have kept the columns intact, correcting only grammar and internal references. I did change the title of one column to "The toughest job—Poor performers" because people misinterpreted the previous title, "You're fired."

Each column starts with a rant, followed by a root-cause analysis of the problem, and ending with suggested improvements. I love word play, alliteration, and pop culture references, so the columns are full of them. In particular, most of the column titles and subheadings are either direct references or takeoffs on lyrics, movie quotes, and famous sayings. Yes, I humor myself, but it's part of the fun and outright catharsis of writing these columns. Enjoy!

How Microsoft Is Organized

Because these columns were originally written for an internal Microsoft audience, I thought a short peek inside Microsoft would be helpful.

Currently, product development at Microsoft is divided into seven business divisions, which correspond to our major product areas—Windows, Office, Windows Phone, Interactive Entertainment (including Xbox), Server & Tools (including Windows Server and Visual Studio), Dynamics, and Online Services (including Bing and MSN).

Each division contains roughly 20 independent product units or triads. The groups within the divisions typically share source control, build, setup, work-item tracking, and project coordination, including value proposition, milestone scheduling, release management, and sustained engineering. Beyond these coordinating services, the product units or triads have broad autonomy to make their own product, process, and people decisions.

A typical triad has three engineering discipline managers: a group program manager (GPM), a development manager, and a test manager. A product unit has these three discipline managers report to a product unit manager (PUM). Without a PUM, the triad managers report within their disciplines to directors and eventually to the division president. The other engineering disciplines—such as user experience, content publishing (for content such as online help), build, and operations—might report into the product unit or be shared by the division.

People reporting into the discipline managers work on individual features by forming virtual teams, called *feature teams*, made up of one or more representatives from each discipline. Some feature teams choose to use Agile methods, some follow a Lean model, some follow traditional software engineering models, and some mix and match.

How does Microsoft keep all this diversity and autonomy working effectively and efficiently toward a shared goal? That’s the role of the division’s shared project coordination. For example, the division value proposition sets and aligns what the key scenarios, quality metrics, and tenets will be for all triads and their feature teams.

Sample Tools and Documents

The sample tools and documents identified in this book as Online Materials can be downloaded from the following page:

http://go.microsoft.com/FWLink/?Linkid=220641

Table of Online Materials

Tools	Column	Chapter
SprintBacklogExample.xls; SprintBacklogTemplate.xlt	"The Agile bullet"	2
ProductBacklogExample.xls; ProductBacklogTemplate.xlt	"The Agile bullet"	2
SpecTemplate.doc; SpecChecklist.doc	"Bad specs: Who is to blame?"	3
InspectionWorksheetExample.xls; InspectionWorksheetTemplate.xlt; Pugh Concept Selection Example.xls	"Review this—Inspections"	5
InterviewRolePlaying.doc	"Out of the interview loop"	9

System Requirements

The tools provided are in Microsoft Office Excel 2003 and Microsoft Office Word 2003 formats. The basic requirement for using the files is to have Word Viewer and Excel Viewer installed on your computer. You can download both viewers from:

http://www.microsoft.com/downloads/en/details.aspx?familyid=941b3470-3ae9-4aee-8f43-c6bb74cd1466&displaylang=en.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

http://go.microsoft.com/FWLink/?Linkid=220642.

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*

Chapter 1

Project Mismanagement

In this chapter:

My first column was published in the June 2001 issue of the Microsoft internal webzine, "Interface." I wanted a topic that truly irked me, in order to get into the character of I. M. Wright. Work scheduling and tracking was perfect.

The great myths of project management still drive me crazy more than any other topic:

1. *People can hit dates (projects can hit dates, but people can't hit dates any better than they can hit curveballs).*
2. *Experienced people estimate dates better (they estimate work better, not dates).*
3. *People must hit dates for projects to hit dates (people can't hit dates, so if you want your project to hit dates you must manage risk, scope, and communications, which mitigate the frailty of human beings).*

In this chapter, I. M. Wright talks about how to manage risk, scope, and communications so that your projects are completed on time. The first two columns are specifically about scheduling, followed by columns on managing late issues (what we call "bug triage"), death marches, lying to cover issues, quick and accurate estimation, managing services, managing risk, and coordinating large projects that might use a mix of methodologies.

One last note: a great insight I've gained from many years at Microsoft is that project management happens differently at different levels of scale and abstraction. There is the team or feature level (around 10 people), the project level (between

50 and 5,000 people working on a specific release), and the product level (multiple releases led by executives). Agile methods work beautifully at the team level; formal methods work beautifully at the project level; and long-term strategic planning methods work beautifully at the product level. However, people rarely work at multiple levels at once; in fact, years typically separate those experiences for individuals. So people think effective methods at one level should be applied to others, which is how tragedies are often born. The moral is: small tight groups work differently than large disjointed organizations. Choose your methods accordingly.

—Eric

June 1, 2001: "Dev schedules, flying pigs, and other fantasies"

A horse walks into a bar and says, "I can code that feature in two days." Dev costing and scheduling is a joke. People who truly believe such nonsense and depend on it are fools, or green PMs. It's not just an inexact science; it's a fabrication. Sure there are people out there who believe that coding can be refined to a reproducible process with predictable schedules and quality, but then my son still believes in the tooth fairy. The truth is that unless you are coding something that's 10 lines long or is copied directly from previous work you have no idea how long it is going to take.

Eric Aside Program Managers (PMs) are responsible for specifying the end user experience and tracking the overall project schedule, among other duties. They are often seen by developers as a necessary evil and thus are given little respect. That's a shame because being a PM is a difficult job to do well. Nonetheless, PMs are a fun and easy target for Mr. Wright.

Richter-scale estimating

Sure, you can estimate, but estimates come on a log scale. There's stuff that takes months, stuff that takes weeks, stuff that takes days, stuff that takes hours, and stuff that takes minutes. When I work with my GPM to schedule a project, we use the "hard/medium/easy" scale for each feature. *Hard* means a full dev for a full milestone. *Medium* means a full dev for two to three weeks. *Easy* means a full dev for two to three days. There are no in-betweens, no hard schedules. Why? Because we've both been around long enough to know better.

In my mind, there are no dates for features on a dev schedule beyond the project dates—milestones, betas, and release. A good dev schedule works differently. A good dev schedule simply lists the features to be implemented in each milestone. The "must-have" features go in the first milestone and usually fill it. Fill is based on the number of devs and the "hard/medium/easy" scale. The "like-to-have" features go in the second milestone. The "wish" features go in the third milestone. Everything else gets cut. You usually don't cut the "wish" features and half of the "like-to-have" features until the second week of the third milestone when everyone panics.

Eric Aside Milestones vary from team to team and product to product. Typically, they range from 6 to 12 weeks each. They are considered project dates that organizations (50–5,000 people) use to synchronize their work and review project plans. Individual teams (3–10 people) might use their own methods to track detailed work within milestones, such as simple work item lists, product backlogs, and burn-down charts.

Risk management

This brings me to my main point. Dev costing and scheduling is not about dates or time. It is about risk—managing risk. We ship software, whether it's a packaged product or web service, to deliver the features and functionality that will delight our customers. The risk is that we won't deliver the right features with the right quality at the right time.

A good dev schedule manages this risk by putting the critical features first—the minimum required to delight our customers. The "hard/medium/easy" scale determines what is realistic to include in that minimal set. The rest of the features are added in order of priority and coherency.

Then you code and watch for features that go from harder to easier and from easier to harder. You shuffle resources to reduce your risk of not shipping your "must-have" features with high quality in time. Everything else is gravy and a great source of challenging but non-essential projects for interns.

Eric Aside The irony is that while almost every engineer and manager agrees with ordering "must-have" features first, few actually follow that advice because "must-have" features are often boring. They are features such as setup, build, backward compatibility, performance, and test suites. Yet you can't ship without them, so products often slip because of issues in these areas.

It is so important to shoot down the "feature dates" myth because devs working to meet feature dates undermine risk management. The only dates that count are project dates, milestones, betas, etc.—not feature dates. Project dates are widely separated, and there are few of them. They are much easier to manage around. If devs believe they must meet a date for a feature, they won't tell you when they are behind. "I'll just work harder or later, eh heh, eh heh."

Meanwhile, you are trying to manage risk. One of your risk factors is an overworked staff. Another is a hurried, poor-quality feature. Another is losing weeks of time when you could have had two or three devs or more senior devs working on a tough issue. You lose that time when your dev staff thinks their reviews revolve around hitting feature dates instead of helping you manage the risk to the product's critical features.

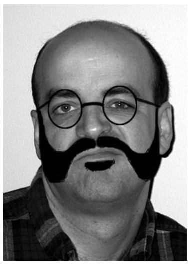
The customer wins

When you make it clear to your dev team that the success of the product depends on your ability to manage the risk to critical features, everything changes. Sure, getting extra features is a nice bonus, but the key is the focus on communicating risk areas and working together to mitigate them.

When everyone understands the goal, everyone works better to achieve it. This also helps to boost morale when the tough cuts are made, and it rewards mature decisions by junior staff. In the end, our customers are the big winners because they get the features they really want with the quality they expect, instead of the features that happened to make it at whatever level of quality sufficed.

BTW, everything I said about dev scheduling applies equally well to test scheduling.

October 1, 2001: "Pushing the envelopes: Continued contention over dev schedules"



Time to reply to comments about my June column: "Dev schedules, flying pigs, and other fantasies." Most comments were quite flattering, but I won't bore you with just how right I am. Instead, allow me to address the ignorant, incessant ramblings of the unenlightened, yet effusive, readers of this column.

Eric Aside This was my first and only "mail bag" column, with responses to e-mail I received. I continue to get plenty of "feedback" on my column, but once the column became popular the number of new topic requests vastly outweighed the value of answering e-mail on a past topic. However, looking back over this early column makes me wonder if Mr. Wright should empty the mail bag again.

Software engineering is clearly ambiguous

I am incredulous at the supposition that development of a feature cannot and should not be scheduled. The statements in the article accurately portray the activity of "coding." Unfortunately, this is what Jr. High schoolers do when they are throwing together a VB app to decode messages to each other. We, on the other hand, are supposed to be software engineers and not hackers.

—Incredulous ignoramus

I hear this kind of thing often, and it just needs to stop. Bank managers don't manage banks and software engineers don't engineer software. They write software, custom software, usually from scratch, with no prior known measures of nominal operating range, tolerances, failure rates, or stress conditions. Sure, we have those for systems, but not for coding itself.

I went to an engineering school. Many of my friends were electrical, civil, aeronautical, or mechanical engineers. Engineers work on projects in which the building blocks and construction process are well defined, refined, and predictable. While there is great creativity in putting the building blocks together in novel ways to achieve an elegant design for a custom configuration, even the most unusual constructions fall within the tolerances and rigor of known qualities and behaviors.

The same cannot be said for software development, although many are trying to reach this goal. The building blocks of software are too low level and varied. Their interactions with each other are too unpredictable. The complexities of large software systems—such as Windows, Office, Visual Studio, and the core MSN properties—are so far beyond the normal scope of engineering that it is beyond hope to make even gross estimates on things like mean-time-to-failure of even small function changes in those systems.

So for better or worse, it's time to get past wishful thinking and high ideals and return to reality. We've got to accept that we are developers, not engineers. We simply cannot expect the predictability that comes with hundreds or even thousands of years of experience in more traditional engineering any more than we can expect a computer to do what we want instead of what we tell it. We just aren't there yet.

Eric Aside Now, six years after I wrote this column, Microsoft measures mean-time-to-failure of much of our software. In addition, methods are becoming available to treat programming as engineering, which I describe in the later column, "A software odyssey—From craft to engineering," in Chapter 5. Even so, I stand by this column as an accurate reflection of software development as a field that has grown past its infancy but remains in its teenage years as compared to its fully grown engineering brethren.

Believe half of what you see and none of what you hear

If I'm relying on another team/product group for a feature or piece of code, I sure don't want to hear, "It should be done in this milestone." I want dates. I need specifics.

—In need of a date

I could write several columns on dependencies and component teams, and perhaps I will, but for now I'll just discuss dependency dev schedules. First of all, if your dependency did have a dev schedule, would you believe it? If you said, "Sure, what choice do I have?" start taking Pepcid now before your ulcer develops. It's not only the dev schedule either. Don't believe anything dependencies say—ever. If they are in the next room and tell you it's raining, check your window first.

This doesn't mean you can't work with dependencies—you can, and it can be a great experience and a windfall for your team, product, and customers. You just must keep a close eye on what's happening. Get regular drops and conduct automated testing of those drops. Get their read/write RAID RDQs and watch their counts and problem areas. Send your PM to their triage meetings. Get on their e-mail aliases.

Eric Aside Check the glossary for help with these bug-tracking references.

Basically, watch dependencies like a hawk; they are an extension of your team and your product. The more you stay in touch and current, the better you will be able to account for shortcomings and affect changes. As for when features will be ready, you simply must rely on your influence to up priorities and on your communication channels and private testing to know when features are *really* ready.

Motivation: It's not just pizza and beer

Your general sentiments make more sense for early level planning of a project than the final milestone before shipping. You need to address issues such as how schedules are often used as management tools to drive performance of the team, providing deadlines and time constraints to execute against.

—Can't find the gas pedal

First, let me reiterate, if you hold devs to features dates, they will lie and cheat to meet the dates. They will lie about status, and they will cheat on quality and completeness. If you don't want to experience either of these from your dev team, you need to come up with a better motivational mechanism. I've used three different approaches in coordination with each other to great effect.

First, at a basic level, there are the Richter-scale estimates themselves. My devs know that I expect each feature to be done in roughly that amount of time. If a two-week task takes two and a half weeks, that's probably okay. If it's taking much longer, there's usually a good reason and the dev will let me know. The lack of a good reason provides ample motivation. However, because there's no hard date, lying and cheating are rare.

The second motivational tool is finishing the milestone. This can be dangerous in that it can invite shortcuts, but the overall effect is to encourage devs to work hard from the start and to know when they are behind. The key difference between a feature date and a milestone date is that the latter is a team date. The whole team works together to hit it. Therefore, there is less individual pressure to cut corners. However, that still can happen, which leads me to the last and most effective technique.

Eric Aside This notion of a self-directed team working toward a clearly defined common goal is central to many agile techniques, though back in 2001 I didn't know it.

The last motivational tool that I use is by far the best. I make it clear to the team which features are the must-ship features, the ones we must finish first. I tell them that everything else can and will be cut if necessary. Unfortunately, the must-ship features are often among the most mundane to code and the least interesting to brag about. So I tell my team that if they want to work on the cool features, they must first complete and stabilize the critical features. Then they will be rewarded by working on the less critical and far flashier stuff. This kind of motivation is positive, constructive, and extremely effective. Works every time.

Sinking on a date

Continued from the previous quote: [You also need to address] that schedules are an absolute necessity for aligning the work of different functional areas (not just Dev, but PM, QA, UE, Marketing, external partners).

—Brain out of alignment

If you really needed solid feature dates to synchronize disciplines and dependencies, no software would ever ship. Of course, we do ship software all the time—we even shipped a huge effort, Office XP, on the exact date planned two years in advance. Thus, something else must be the key.

What really matters is agreeing on order, cost, and method, and then providing timely status reports. The agreements should be negotiated across the disciplines, and the process for giving status should be well defined and should avoid blocking work.

- **Order** Negotiating the order of work on features is nothing new, although there are some groups who never agree on priorities.

- **Cost** Negotiating cost is often done between the dev and PM. (For example, a dev says, "If we use a standard control, it'll save you two weeks.") But sometimes it's left just to the dev. It should also include test and ops.
- **Method** Negotiating the methods to be used is frequently done for PM specs, but it's done less frequently for dev and test specs—to their detriment.
- **Status reporting** As for timely reporting of status, you really need check-in mail and/or test release documents (TRDs) to keep PM, test, and ops aware of progress. Test needs to use alerts for blocking bugs. And PM should use something like spec change requests (SCRs) to report spec changes. (To learn more about SCRs, read "Late specs: Fact of life or genetic defect?" in Chapter 3.)

If the different groups can plan the order of their work, know about how long it will take, have confidence in the methods used, and maintain up-to-date status reports, projects hum. Problems are found, risk is mitigated, and surprises are few. More importantly, no one is pressured to do the wrong thing by artificial dates. Instead, everyone works toward the same goal—shipping a delightful experience to our customers.

May 1, 2002: "Are we having fun yet? The joy of triage."

Tell me if I don't have this concept nailed…

Program managers want an infinite number of features in zero time, testers and service operations staff want zero features over infinite time, and developers just want to be left alone to code cool stuff. Now, put the leads of each of these disciplines with their conflicting goals in the same room, shut the door, and give them something to fight over. What happens? Triage!

Eric Aside As product development issues arise (such as incomplete work items, bugs, and design changes), they are tracked in a work item database. Triage meetings are held to prioritize the issues and decide how each will be addressed. This can be a source of conflict (understatement).

It's amazing that blood doesn't start leaking out from under the triage room door. Of course, that's what solvents are for. But does it have to be a bloodbath? Most triage sessions are certainly set up that way. Some of the most violent arguments I've seen at Microsoft have happened behind the triage door. Is this bad, or is it "by design"?

War is hell

As anyone who's been through a brutal triage can tell you, it's not good. Rough triages leave you battered and exhausted even if you win most of the arguments.

Basically, dysfunctional triages go hand in hand with dysfunctional teams. They generate bad blood between team members and often set a course of reprisals and unconstructive behavior.

Why should this be? We encourage passion around here. We want people to fight for what they believe and to make the right decisions for our customers. What’s wrong with a little healthy competition? Well, when it’s not little and it’s not healthy, it’s not good.

It’s nothing personal

Bugs shouldn’t be considered personal, but they are.

- To the tester who found it, the bug represents the quality of his labor: “What do you mean the bug isn’t good enough to fix?”
- To the program manager who wrote the feature, the bug represents a challenge to her design: “It breaks the whole idea of the feature!”
- To the service ops staff, the bug represents real and continuing work: “Yeah, you don’t care about the bug; you’re not the one who’s going to have to come in at 3:00 A.M. to reboot the server!”

> **Eric Aside** Interesting note here about 3:00 A.M. reboots. Like most software service companies, Microsoft is now moving away from service operations being on call 24/7. Instead, we are designing services to automatically heal themselves (retry, restart, reboot, reimage, replace). Service operations people, working regular business hours, simply swap components on the automatically generated replacement list.

- To the developer, the bug represents a personal value judgment: “It’s not that bad.”

Triage decisions should be based on doing what is right for our customers and for Microsoft, not on personal feelings. Yet, because of the personal investment that each discipline places on bugs, triage discussions get off track in a heartbeat.

Five golden rules of triage

How can you keep triage on track and constructive? Follow my five golden rules of triage:

1. **Shut the door.** Triage is a negotiation process, and negotiations are best held in private. It is far easier to compromise, to bargain, and to be candid when the decision-making process is confidential. It also allows the triage team members to present their decisions as team decisions.
2. **All decisions are team decisions.** After a consensus is reached, it is no longer the decision of individuals, but of the group. Everyone stands behind the choices as a

team—with no qualifications. A triage team member should be able to defend every decision as if it were her own.

3. **Just one representative per discipline.** Triage must be decisive. Unfortunately, the more people involved, the longer the process; the more personal feelings, the more difficult it is to reach a conclusion. A single individual can make a decision the fastest, but you need the viewpoints of each discipline to make an informed choice. So the best compromise between decisiveness and discipline perspective is reached through having one representative per discipline.
4. **One person is designated to have the final say.** If the team can't reach consensus, you need someone to make the call—ideally, this never happens. Personally, I prefer the PM to have the final say because PMs are used to collaboration and realize the consequences dictating decisions. They tend not to abuse the privilege. However, the very threat that someone from another discipline (let alone the PM!) could impose his decision on the team is enough to drive people to consensus.
5. **All decisions are by "Quaker" consensus.** This is the most important rule. Regular consensus implies that everyone agrees, but that bar is too high to meet for something as difficult and personal as triage. "Quaker" consensus means that no one objects—the team must work toward solutions that everyone can live with. This presents a far more achievable and often more optimal outcome. (Note that "Quaker" simply refers to the people who came up with this notion; it has no religious significance.)

Follow these five rules and your triage will become more cordial, constructive, and efficient. However, there are some subtleties that are worth fleshing out.

The devil is in the details

Here are a few more details that can help your triage run more smoothly:

- If your arguments are about people instead of bugs, change the focus to what's best for the customer and the long-term stock price. This perspective takes personal issues out of the discussion and puts the focus where it should be.

> **Eric Aside** Throughout the columns, I talk about focusing on the customer and the business, instead of on personal issues. You might wonder why you shouldn't just think about the customer and leave the long-term stock price out of it. I'm sympathetic to this point of view, but I also know that we don't get to serve the customer if we are no longer in business. It helps to have a business plan that aligns our work to provide sustainable benefits to our customers.

- If you need extra information about a bug or a fix, it's sometimes necessary to invite someone from outside the triage team to join you, either by phone or in person. Always complete your questioning and bid them farewell before you begin debating your decision. Otherwise, confidentiality is broken and the decision may cease to be a triage decision.
- If you'd like to teach a member of your team about the triage process, invite him to join a triage session, but instruct him to be a fly on the wall during discussions and stress the confidential nature of the negotiating process.

It's hard to let go, isn't it?

If one or more of the triage members can't seem to let go of an issue, give them a small number of "silver bullets." The rule behind silver bullets is that you can use them at any time to get your way, but when they are used, they are gone. When a person won't give in on an issue ask, "Do you want to use one of your silver bullets?" If so, the team is bound to support the decision. Usually the person will say, "Uh, no it's not that important," and the team can move on.

Eric Aside This triage column has produced a significant amount of controversy over the years, particularly this paragraph about "silver bullets." Some complain about using the term "bullet" instead of "token," but the primary complaint is that a critical team decision could be made by an individual using his "silver bullet." In practice, this never happens. Silver bullets help people prioritize by associating importance with a scarce resource. People who don't need the help don't use their supply. Thus, if someone abused a silver bullet on a critical issue, there's always someone else with spare tokens to counter. That said, I've never heard of this happening.

Finally, when it comes to resolving the triaged bugs in a database:

- Always use the "Triage" label to indicate that this was a triage decision.
- Always explain the thinking behind the triage team's decisions.
- Never resolve a bug (especially external bugs) unless that's the last time you want to see it. Too often, teams resolve ship-blocking bugs as "external" or "postponed" when what they mean is, "We don't want to deal with this bug now, we'll deal with it later." But because the bug is "resolved," it falls off the "active" radar and the issue gets lost.

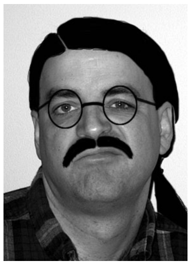
Eric Aside You can find my column on bug fields, priorities, and resolution values, called "Am I bugging you? Bug Reports," in Chapter 2.

Take care of the little things

Triage is arguably one of the most important duties that you perform as a team. Triage health almost always directly corresponds to the health of the project and of the group. The real beauty of this relationship is that making triage sessions more positive, productive, and pleasant usually leads to the same change in your work and your team. But fixing triage issues is much easier and involves fewer people than fixing entire team and project issues.

The best thing about improving your team's triage sessions is that when you get it right, it can be the most fun that you have all day. When triage focuses on bugs instead of people and consensus instead of carnage, the stress of the exercise comes out as humor instead of aggression and frustration. Teams working well together often have triages that are filled with wisecracks, inside jokes, twisted ironies, and hilarious misstatements. Make the right adjustments to your triage techniques, and the laughter may be echoing down the halls. Better keep the door shut.

December 1, 2004: "Marching to death"



Ever been in a project death march? Perhaps you are in one now. There are many definitions of such projects. It basically comes down to having far too much to do in far too little time, so you are asked to work long hours for a long time to make up the difference. Death marches get their name from their length, effort, and the toll they take on the participants. (I apologize for how insulting this is to those whose relatives experienced actual death marches in WWII; but unfortunately, software is full of insensitive word usage.)

It's hard to fathom why groups continue to employ death marches, given that they are almost certain to fail, sometimes spectacularly. After all, by definition you are marching to death. The allure escapes me.

Stabs in the dark

Inept management continues to engage in death marches, so I'll take a few stabs at explaining why.

Eric Aside Death marches are hardly unique to Microsoft, nor are they pervasive at Microsoft, a fact I learned much to my surprise when I joined the company. Microsoft's reputation for long hours preceded it when I joined the company in 1995. I was concerned because I had a two-year-old boy and another child in the works, but my boss assured me that death marches were not the rule. His word was true, yet there are isolated instances when management at Microsoft and other companies still resort to this inane and arcane practice.

- **Management is remarkably stupid.** Managers choose to act without thinking about the consequences. They take a simpleton’s approach: Too much work to do? Work harder. At least managers can say they’re doing something, even if it is probably wrong.
- **Management is incredibly naïve.** Managers don’t know that a death march is doomed to fail. Somehow they were either asleep for the last 25 years or never read a book, article, or website. They assume that adding at least four hours a day and two days a week will double productivity. The math works out—unfortunately, humans aren’t linear.
- **Management is tragically foolish.** Managers think that their team will be the one to overcome the insurmountable odds. Rules and records were meant to be broken. They’ve got the best team in the world, and their team will rise to the challenge. Apparently, they see no difference between outrunning a bull (hard) and outrunning a bullet (impossible).
- **Management is unconscionably irresponsible.** Managers know that a death march will fail, destroying their team in the process; but they do it anyway in an effort to be worshiped as heroes. Managers reward this behavior with free meals, gold stars, and high ratings, knowing that our customers and partners won’t be screwed by the garbage we deliver until after the next review period. I think these managers are the most deserving of a verbal pummeling by Steve’s staff.

> **Eric Aside** *Steve* refers to Steve Ballmer, our beloved Chief Executive Officer, who is a strong advocate for work-life balance and practices it himself. I’ve met him several times while he was cheering on his son at a basketball game or going out to a movie with his wife.

- **Management is unaccountably spineless.** Managers know that the death march is doomed, but they lack the courage to say “no.” Because they won’t be held responsible if they follow the herd, there is little consequence for these cowards. Sure, the project will fail and their employees will hate them and leave, but at least they’ll have war stories to share with their gutless, pathetic pals.

Many people have written about the ineffectiveness of software project death marches, but somehow the practice continues. I can’t reason with the foolish and irresponsible, but I can enlighten the stupid and naïve and give alternatives to the spineless.

A litany of failure

Some enlightenment for the ignorant: Death marches fail because they…

- **Are set up for failure.** By definition you have far too much to do in far too little time. Of course you fail.
- **Encourage people to take shortcuts.** Nothing could be more natural than to find cheap ways to leave out work when you are under pressure. Unfortunately, shortcuts lower quality and add risk. That may be okay for small items and short time periods. But those risks and poor quality bite you when the project drags on.
- **Don't give you time to think.** Projects need slack time to be effective. People need time to think, read, and discuss. Without that time, only your first guess is applied. First guesses are often wrong, causing poor design, planning, and quality, and leading to dramatic rework or catastrophic defects later.
- **Don't give you time to communicate.** You could make a good argument that miscommunication and misunderstanding are at the root of all evil. Even good projects commonly fail because of poor communication. When people don't have spare time and work long hours, they communicate less and with less effectiveness. The level of miscommunication becomes an insurmountable obstacle.
- **Create tension, stress, and dysfunction.** Congeniality is the first to go when the pressure is on. Issues become personal. Accidents get amplified and misconstrued. Voices get raised, or even worse, people stop talking.
- **Demoralize and decimate the workforce.** All the bitterness, all the tension, and all the long hours away from family and friends take their tolls on the psyche and relationships. When the project inevitably fails to meet its dates and quality goals, people often snap. If you're lucky, it just means switching groups at the end of the project. If you're unlucky, it means leaving the company, divorce, health issues, or even life-threatening addictions.

By the way, managers often confuse the long hours some employees ordinarily put in with death marches. Death marches are an entirely different dynamic. The difference is that a death march forces you to put in those hours. When people voluntarily put in long hours, it's often because they love it. Such hours are full of slack time. There isn't any tension or cause for taking shortcuts.

Eric Aside This is a critical point people often miss. Voluntary long hours are completely different from death marches.

- **Undermine confidence in the process.** It doesn't take a genius to realize that death marches are a response to something going wrong. The message this sends to our employees, customers, and partners isn't dedication, it's incompetence. Avoiding the real issues and just working harder only undermine our corporate standing further.
- **Don't solve the problem.** Working longer hours doesn't solve the underlying problem that caused the project team to have far too much to do in far too little time. Until the underlying problem is solved, no one should expect the project to do anything but get worse.
- **Reduce your options.** When you've taken shortcuts, introduced poor designs and plans, created dramatic rework and defects, randomized your messaging, encouraged people to slit each other's throats, demoralized the staff, undermined confidence in our ability to deliver, and still failed to hit dates and quality goals—leaving all the original issues unresolved—you have few options left. Usually this leads to dropping the quality bar, slipping the schedule, and continuing the death march. Nice job.

The turning point

So, if you find yourself with far too much to do in far too little time, what should you do? On a practical level, the answer is remarkably easy. Figure out why you've got far too much to do and far too little time to do it.

The answer isn't, "Because those are the dates and requirements from management." Why are those the dates and requirements from management? What would management do if you didn't hit certain dates or certain requirements? Would they slip the schedule? How much? Would they cut? Which features? Are there more fundamental changes you could make in the process or approach that would alter the dynamic? Tell management that your goal is to hit the dates and requirements, but you have to plan for the worst case.

Then plan for the worst case. Build a plan that hits the worst acceptable dates with the least acceptable features. If you are still left with too much to do for the available time, raise the general alarm. Your project is dead in the water. If the worst-case plan is perfectly achievable, focus all your efforts on achieving it. Message to your employees that doing more means a review score of 3.5+, but doing less means a score below a 3.0.

Eric Aside The numbers refer to the old Microsoft rating system, which ranged from 2.5 to 4.5 (the higher the rating, the better the rewards). While a 3.0 was acceptable, most people pursued and received a 3.5 or higher.

The road less traveled

What you've done is escaped from the death march and created slack time to improve. Your team will likely go far beyond the minimum, but they will do so without taking shortcuts, making poor decisions, or engaging in cannibalism. You will deliver what's needed on time and build confidence with your partners and customers.

As reasonable as this sounds, it is hard to do on an emotional level. Planning for the worst case feels like giving up. It feels weak and cowardly—like you can't handle a challenge. How ironic; in actuality, it is entirely the opposite.

Not facing the crisis is weak and cowardly. Pretending the worst won't happen is deceitful and irresponsible. Show some guts. Face the facts. Be smart and save your partners, customers, and employees from the anguish at the end of the road. Come out on the other side with value delivered and with your team, your life, and your pride intact.

Eric Aside On a recent nine-month project, my team had a critical service dependency take a three-month slip toward the end of the project. My team went from having four months to complete a major feature to one month. We could not slip the schedule, and we could not cut the feature (both were already committed to partners). We didn't go through a death march. Instead, we moved into our dependency's development environment and worked in parallel as they completed their service. This strategy not only recovered time but also reduced rework since we were able to give feedback to the service team on very early builds. We shipped on time with great customer reviews. It was difficult and people did work hard, but they also took time off and were pressured only by the desire to ship a high-quality product and support their teammates. We retained everyone after release.

October 1, 2005: "To tell the truth"

I cannot tell a lie—catchy phrase, but a children's tale. Everybody lies from time to time. Sometimes it's strategically leaving out details. Sometimes it's not saying how you truly feel. Sometimes it's an out-and-out fabrication. No matter the reason or circumstance—lying is deception, pure and simple.

Some might rationalize this behavior as "white lies," but it amounts to the same thing: dishonesty. If someone catches me lying, no matter how slight, I fess up immediately, sincerely, and remorsefully. When I was a kid, I would perpetuate and cover up the deception. But I've since learned that covering it up is far more damaging than the original offense. Most people, including me, aren't lying to offend anyone; our motivation is pure expediency.

Therein lies the core truth: deception is basically a quick and dirty way to avoid a problem. How is this relevant to software development? Because by focusing on "when" and "why" you or your team lie, you can pinpoint everything from quality issues to retention troubles to increased productivity.

Suffer from delusions

Lying is one of a handful of valuable process canaries that can warn you of trouble. Why? Because lying, cycle time, work in progress, and irreplaceable people hide problems. Long cycle times and large amounts of work in progress hide workflow difficulties. Irreplaceable people hide tool, training, and repeatability problems. Lying can hide just about anything. Scrutinizing these process canaries exposes the problems and enables improvement.

Eric Aside I write about each of these process canaries in other columns: "Lean: More than good pastrami" in Chapter 2, and "Go with the flow—Retention and turnover" in Chapter 9. As for the five whys, like Lean, that concept comes from Toyota.

The key is getting to the root cause of the lie. One of the best ways to do this is to apply "The five whys"—that is, ask "why" five times:

- Why you are lying? What pain are you hiding from?
- Why hide from that pain? What's the danger?
- Why would that happen? Is there a way to mitigate the danger?
- Why aren't you mitigating the danger already? What actions do you need to take?
- Why are you just sitting there? Act!

To practice applying these ideas, let's go over some common examples of lying at work. We'll apply the five whys to uncover the root cause and discuss how to fix it. Here are our foul foursome of falsehoods:

- Perverting the meaning of the word "Done"
- Weaseling out of a tough review message
- Face-lifting progress reports for your clients and boss
- Denying rumors about a reorganization

Put a fork in me

Say your dev team is supposed to finish up feature development on Monday. On Monday, you go through the team and everyone says, "I'm done." Later, you find that more than half the features are full of bugs and a quarter don't handle error conditions, accessibility, or stress. You could ask, "Why does my team stink?" But the better question is, "Why did my team lie?" Let's ask the five whys:

- **Why did my team lie about being done; what are they hiding from?** They had a deadline to meet, and not meeting it would drop their standing within the team. The criterion for meeting the deadline was simply saying they were done.
- **Why just say you're done and not mean it—what's the danger?** No one wants to look bad. Unfortunately, there was no personal danger to saying, "I'm done." So why wouldn't they lie? The danger was to the team. That's the real problem.
- **Why would that happen; can you mitigate it?** There was no verifiable team definition for "done." This opened the door to deception. To mitigate it, you need a clear definition, accepted by the team, with an objective means of verifying it has been met.
- **Why don't you have a clear definition of "done"? What more do you need?** When you agree on a definition and means of verification, you need to put the tools in place. Say the definition is 60% unit test coverage with 95% of tests passing, along with a three-peer code inspection that finds 80% of the bugs. Now you need to add code coverage and a test harness to your build for the unit tests, as well as an inspection process with the appropriate time scheduled for the inspectors and inspections.
- **Why are you sitting there?** Most of what you need is in Toolbox—aside from the nerve to challenge the meaning of "done" in the first place. The key is to focus on the cause of the deception, and then rectify the root of the problem.

Eric Aside Toolbox is a Microsoft internal repository for shared tools and code. It holds tools that measure code coverage, run unit tests, and even calculate bug yields for code inspections. Many of these internal tools make their way into Visual Studio, Office Online Templates, and other shipping products.

Give me a straight answer

You manage a 4.0 performer you really value, and you've told her so. Your division runs a calibration meeting, and your 4.0 performer drops to a 3.5 relative to her peers in the division. It's easy to say to your employee, "Well, I thought you deserved a 4.0, but as you know, the review system is relative and I can't always give you the rating you deserve."

You're lying, not because what you are saying isn't true, but because you're leaving out your role in the process. Again, let's cover the five whys:

- **Why leave out your responsibility; what are you hiding from?** You like the employee and don't want to be blamed.

- **Why hide from blame; what's the danger?** Your employee might not like you and may leave the team.
- **Why would that happen; can you mitigate it?** You are the messenger, your employee feels helpless, and you are no help. You can mitigate the impact by telling your employee how to get the review score she wants.
- **Why aren't you already telling her; what more do you need?** You need to know why she got the 3.5 instead of the people who were awarded 4.0.

> **Eric Aside** The process around differentiated pay based on performance is a common source of complaints across the high technology industry. Like the numerical rating system, we've changed the process many times at Microsoft, but it's always been about comparing your work to the work of others doing the same job at the same level of responsibility. What managers should always do is understand and clearly articulate how their employees can improve to compare more favorably.

- **Why are you sitting there?** Find out what differentiated the 4.0 from the 3.5 performers, and then tell your employee. She'll have clear guidance on how to improve and be in control of that improvement. Sure, she'll still be unhappy, but at least you helped her and she can do something about it.

Lipstick on a pig

Your team is falling behind on the schedule. You've got a ton of bugs and can't keep up. Your clients and boss demand to know the status. Instead of a fair representation, you paint a rosy picture in the hope that your team will be left alone long enough to catch up. Aside from feeling bad about being a gutless slimeball, what should you do? Here are the five whys:

- **Why the desperate move; what are you hiding from?** You don't want to look bad or have others interfere.
- **Why hide from blame; what's the danger?** You're afraid your project will get cut or transferred to someone else because of your perceived incompetence.
- **Why would that happen; can you mitigate it?** If your clients and boss get blindsided by your team slipping, they won't trust you to take care of it. You can mitigate the problem by being transparent so that no one gets surprised, and by having a solid plan to get on track, which earns you the confidence of your clients and boss.
- **Why aren't you already transparent; what more do you need?** It's a ton of work to constantly collect status from your team and post it or send e-mail. Instead, post your schedule and bug data directly on your SharePoint site, warts and all. Have your team update it directly, right there for the world to see. Use charts to make progress (or lack thereof) obvious. When it's posted, point your team to it. Everyone will get the picture, and you'll be able to drive a plan to get on track.

- **Why are you sitting there?** None of this is hard. Transparency drives the right behavior. It also drives trust, which really is the key asset to being successful.

Look at all these rumors

Rumors are flying around about another reorg. Your PUM has told you to keep it quiet; but meanwhile, your team is getting randomized. Naturally, when the topic comes up at your team meeting, you deny any knowledge of the reorg; instead, you remind folks of the evil of rumors and that the team needs to focus on their deliverables. However, you are overcome with guilt, dreading the day when your whole team realizes that you lied to their faces.

Eric Aside A Product Unit Manager (PUM) is the first level of multidisciplinary management at Microsoft. PUMs are typically responsible for individual products, such as Excel, that are part of larger product lines, such as Office. PUMs might also be responsible for significant components of larger products, such as DirectX for Windows. Reorganizations, also known as reorgs, typically start at the top levels of management and slowly work their way down over the following 9 to 18 months. I wrote more about reorgs in my column "How I learned to stop worrying and love reorgs," which appears in Chapter 10. PUMs are becoming a rare species at Microsoft as the company moves toward a functional organizational structure, as I describe in "Are we functional?," also in Chapter 10.

- **Why deny the rumors; what are you hiding from?** Basically, your boss told you to deny them. You don't want your team randomized any more than your boss does.
- **Why worry about randomization; what's the danger?** You're concerned your team will get so caught up in the rumors that they'll fail to meet their commitments. In addition, some team members might even leave the group for fear of unwanted changes.
- **Why would that happen; can you mitigate it?** Most team members, particularly the senior ones, know how bad reorgs can sometimes get. However, no one (including you) knows if the reorg will really happen or how a reorg will actually turn out. So your team's concerns are without a strong base in fact.
- **Why is your team still taking the rumors seriously; what more can you do?** In this case, the problem lies squarely with you. You are taking the rumors too seriously, hiding what you know from your team. You should know by now that only roughly one in three planned reorgs actually happens.
- **Why are you sitting there?** The solution is simple and obvious here: tell the truth. "Yeah, I've heard lots of rumors too. We talk about them in our staff meetings. However, the bottom line is that no one knows whether or not there really will be a reorg until it actually happens. Most planned reorgs don't happen, and we're going to look pretty foolish missing our commitments because we were daydreaming."

I want the truth

I make no judgments about whether or not people should always tell the truth. To do so would be hypocritical and lead to awkward situations when my mother-in-law asks what I think about her decorating.

However, we all work for the same company. You shouldn't have to lie to your coworkers about business issues. Lying hides problems that need exposure. If you're feeling the need to lie, ask yourself why. Then ask again until you resolve what the real problem is. People wonder about how they can deliver on the fourth pillar of Trustworthy Computing, "Business Integrity." Well, now you know.

September 1, 2008: "I would estimate"

When I'm discussing challenges with fellow engineers, the first topic that comes up isn't estimation—it's career and people challenges. That's why those issues are so rampant in these rants. However, "How do you generate task estimates?" is always among the top non-moaning-about-your-manager-or-mates topics. After all, estimation is predicting the future. There are so many unknowns and unforeseen issues that it's impossible to provide the accurate estimates demented despots demand. Isn't it? It must be. Right?

Wrong. Estimation is among the most trivial tasks an engineer has to perform on a regular basis. Get over yourself, it is. It's so easy that there are dozens of seemingly different methods that all give you remarkably accurate predictions of completion time. All those methods come down to one simple concept—how long it took last time is how long it will take this time. Nothing could be easier.

Yeah, you've got to understand the work well enough to compare it to previous work, but that isn't too tough either. No, the real challenge isn't task estimation; the real challenge is accepting the estimate. Estimation is easy; acceptance is hard.

Eric Aside There are many consultants, seminars, and training programs on estimation. I'm sure they'd tell you that estimation is tricky and focus on techniques to avoid the many pitfalls. At the end of the day, what really matters is believing the estimate. It's the hardest thing to do, yet it has the most significant impact on accuracy.

No one would accept the program

Let’s pretend for a moment that you actually keep track of how long it takes you in calendar days to perform various tasks. (You do—the information is right there in your e-mail dates.) Let’s further imagine that you provided those previous times as estimates for doing similar tasks today (you’d be quite accurate). What would the reaction be from your project leads and managers? My guess: “Oh come on, you’ve got to be kidding me!”

This is fun, so let’s take it a step further. Let’s say you told your project leads and managers that your estimates were based on hard dates collected from your previous project. What reasons would they give for not believing this hard data? Here’s the big three:

- Last time was different.
- You get faster the second time.
- Weird stuff happened last time.

Let’s break down these feeble fallacies one at time.

It’s a different kind of flying altogether

The first excuse your manager or project lead will have to reject your hard schedule data from the previous project is that the previous project was different. Things have changed. Perhaps the build system and tools have changed, the design change request process changed, the requirements changed, management changed, or perhaps the moon is in a different position relative to Saturn this time.

Out of all those excuses, only two have a small chance of affecting your estimates—the tool and process changes. Every other factor is superfluous with little or no impact to cycle time.

Even the tool and process changes would have to be extreme to noticeably affect the accuracy of your estimates. Tool changes would have to cut end-to-end build times by a factor of five. Process changes would have to reduce the time of weekly activities by days. Otherwise, the impact is just noise in the estimate.

Look, the more the world changes, the more it stays the same. Deal with it.

Eric Aside Let’s say a task takes you two weeks, give or take a day or two. The tool or process change would need to save you at least one full day every two weeks to matter.

I'm getting better

The second excuse to reject your hard schedule data from the previous project is that you get better the second time around. The funny thing is that you do get better the second time around. The problem is that you're not doing the same project (hopefully). The only things that are the same are the tools, process, project scope, and the general task of software engineering.

You should already be well versed in the general task of software engineering, so getting better at the details of the previous project has no impact on the estimates for the next project. Of course, if you're fresh out of school, then the second project will take less time than the first.

If you did change tools and processes, your performance should actually be worse because it will be your first time using them. That's okay if the changes are small or the benefits are big. Just don't kid yourself about the impact.

You do want to compare the current project to a prior project with similar scope. The better the match, the more accurate the estimate. The big differences between estimation techniques are how they produce matches.

Oh no, not again

The final excuses your manager or project lead will have to reject your hard schedule data from the previous project are all the "weird" things that happened last time. There was that unexpected security patch, the feature that was far more complex than anticipated, the reorganization and associated project reset, not to mention the snowstorm, and that earthquake, yeah, the earthquake. There's no way you should count the earthquake!

You count the frigging earthquake. There's always a surprise patch, feature, reorganization, and natural disaster waiting for you over the course of a project. Always. Random events happen, but their impact on the schedule isn't as unpredictable as the events themselves. Thanks to Lyapunov's central limit theorem, their overall impact averages out. However long it took last time is likely to be nearly the same this time. That is, as long as you don't pretend this time will be different.

Same old wine

Okay, we've proven your project leads and managers are in denial. As a result, they force you to make ridiculous estimates you don't believe, only to blame you later for missing them.

We've shown that accurate estimates are almost trivial to make. The big question remains, "How can you turn your trivial and accurate estimates into ones your project leads and managers will believe?"

That's where task hours are so handy. Instead of making your estimates in calendar days, you make them in task hours—the number of hours it would take if there were no earthquakes, e-mail, or bathroom breaks. Without those distractions, your estimates look far smaller and more reasonable, even though they're no different.

Task hour estimates are slightly harder to make because you don't have the data lying around in your inbox. However, you can estimate task hours quickly, easily, and accurately with a simple technique like planning poker (or it's more accurate and sophisticated sibling, Wideband Delphi).

In planning poker, three or more engineers each estimate the same task privately around a table. They all reveal their estimates simultaneously so no one exerts undue influence. If the estimates match, you're done. If they differ, the high and low estimators explain themselves, the group discusses their thinking, and then the process repeats until the estimates agree. The process also surfaces assumptions before they become a problem.

Once you have believable estimates in task hours, the argument isn't about how long the tasks will take. It's about how many hours you spend on task in a week. Even after subtracting vacation, training, and big group meetings, most teams spend less than half of working hours on task. The rest of their time is in meetings, answering e-mail, lunch breaks, and so on. If your project leads and managers don't believe it, simply have the team spend two weeks tracking their hours. The numbers don't lie.

Eric Aside Even after subtracting vacation days, training, off-site meetings, and other planned nontask time, most engineering teams spend only about 42% of their time on task. You can increase time on task by having days or afternoons set aside for no e-mail or meetings; having feature teams co-located and self-directed, which reduces formal meetings, design mistakes, and overall communication time; and by using methods like Scrum Sprints, which increase team focus.

My current team uses "story points" instead of task hours. The concept is simple. You pick a point count for an average-size task—say 8 points. You estimate other tasks relative to the average-size task. Most teams use a chunking size for estimates—mine likes Fibonacci sizes (1, 2, 3, 5, 8, 13, 21, 34, ...). After several weeks, you calculate how many points the team was able to complete per week. That's the team's velocity. You can update that velocity on a running basis and use it to accurately convert story points to calendar days.

Your results may vary

As I mentioned earlier, there's a certain amount of randomness or "variance" that asserts itself over the course of a project. It averages out, but any one estimate has a chance of being off by some standard deviation. That deviation is a percentage; it scales with the size of the estimate. Thus, a two-day estimate will be accurate give or take a few hours, a two-week estimate might be off by a couple of days (in either direction), and a three-month estimate could be off by a couple of weeks.

As long as you avoid being overly optimistic, the randomness will even out. By the end of the project, your project deviation will be about the same as the deviation from any individual task. If you are overly optimistic ("It can't take this long next time!") your deviations will keep adding up, not averaging out.

The point is that there's little point in estimating a two-day task to the minute or a three-month task to the day. You just need order-of-magnitude estimates, like I talked about in my very first column seven year ago, "Dev schedules, flying pigs, and other fantasies" earlier in this chapter.

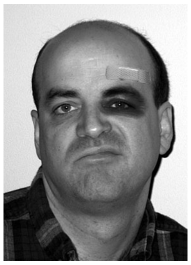
I want to believe

How would you estimate? Focus with your peers on understanding the tasks at hand and their order of magnitude using a technique like planning poker or Wideband Delphi (poker for well-understood tasks, Delphi for others). That's the easy part.

What truly matters in the end is believing and accepting your estimates, then scheduling accordingly. As I've written about before, over-committing is foolish. What's worse is that over-commitment can lead to "Marching to death." (See this column earlier in the chapter.) As I said then, death marches are a strong indicator of weak, cowardly, deceitful, and irresponsible management.

Scheduling trouble is diabolical but completely avoidable. When you prioritize your work properly, putting what's first first, you reduce pressure on your schedule. When you use your estimates to drive realistic commitments, you can deliver reliably to your customers and partners, build trust, and enhance your group's and our company's reputation. Deriving good estimates is easy. Trusting them and yourself is the challenge.

May 1, 2009: "It starts with shipping"



Call me "old school," but I believe in shipping. Trying isn't enough. Getting close isn't enough. Good ideas aren't enough. You've got to ship.

It used to be that Microsoft interviews started with, "What have you shipped?" If you hadn't shipped recently, "Why?" Why? Because you can't deliver customer value if you don't deliver. You can't iterate and improve without finishing an iteration. You can't get customer feedback without customers.

People used to complain that promotions and rewards were disproportionally distributed to those who shipped. I say, "Absolutely, that's how it should be." Does this hurt quality? No, you set a high minimum quality bar and ship. Does it hurt innovation? No, innovators have always risked an initial drop in pay to receive a big payoff should they deliver.

Eric Aside Some people complain that the big payoff doesn't exist at Microsoft for innovative ideas. Those people haven't shipped. The people who successfully ship innovative ideas are the ones who become our organizational and technical leaders.

It all starts with shipping. This is particularly apt with services, where everything literally starts with shipping, and where I'm focusing the rest of this column. Our critics claim that in the new world of services Microsoft has forgotten how to ship. Perhaps, but Microsoft has forgotten more about shipping than most companies will ever know. We just need some reminders and reeducation, especially when it comes to services.

Eric Aside Does a focus on shipping drive death marches? No, death marches delay shipping. As I wrote in "Marching to death," death marches result from a lack of planning and courage. This is particularly important to understand in the services world, where sustainable shipping is critical to long-term success.

Eric Aside By the way, I was criticized for the line "Microsoft has forgotten more about shipping than most companies will ever know." It's arrogant and makes it seem that Microsoft isn't willing to learn new ideas about shipping. Absolutely, I. M. Wright is arrogant and proud of it. As for the claim that Microsoft is unwilling to learn, many of our former competitors would disagree. Microsoft has a track record of learning quickly and iteratively until we overtake. Some might claim it's our size, but we haven't always been big. Some might claim it's our tactics, but tactics aren't enough. You've got to have the right product in the right channel at the right time. That takes tenacity and teaching.

I offer you my service

How much about shipping services has Microsoft forgotten or not get, according to critics? Not as much as they would have you believe, but enough to make you think. Let’s go over the herrings and the heartaches, mixed with a little happiness.

The red herrings:

- Services make you think about everything differently.
- Services center on data while packaged products center on functionality.
- Services have greater security concerns than packaged products.
- Services have serious issues with dependencies.
- Services demand higher quality and faster iterations than packaged products.

The heartaches (and happiness):

- Services run across hundreds of machines, not on a single client.
- Services must scale out automatically.
- Services are easier to switch than packaged products.
- Service upgrades hit everyone instantly.
- Services are living, changing things.

Let’s break these down, starting with the red herrings.

What is that smell?

The first services red herring is a big one, “Services change everything.” As I addressed in “At your service” (in Chapter 6), this is total bovine fertilizer. Services start and end with helping customers achieve their goals, just like all products ever. You focus on the customer experience and what they hope to accomplish or you lose. End of story.

The next three red herrings—centering on data, security concerns, and dependency issues—all apply just as well to shipping packaged products, though it may have taken us longer to realize it. You can’t expose data format changes to customers without chasing them away, on the client or the server. There isn’t a computer product or service today that isn’t vulnerable to attack—you must secure them all. Finally, if you think external dependencies aren’t problematic on the client, you clearly don’t use many drivers. I’m not saying these aren’t real issues—I’m saying they aren’t new or specific to services.

The last red herring is among the most common concerns raised about why shipping services differs from shipping packaged products—high availability and Internet time. Look, it’s not okay for packaged products to never work or require a reboot every time you use them; at least it hasn't been for quite some time. The quality bar is no different for services, though there are plenty of services that fail constantly.

As for Internet time, that hit packaged products a decade ago with the introduction of Windows Update. And if you think that those patches are just security fixes, you haven't been paying attention. More and more we are fixing all kinds of experience issues shortly after customers report them, for services and packaged products. That's a great thing for customers.

However, gradually improving the customer experience every month or every day isn't enough. Both services and packaged products need to ship significant, orchestrated updates to deliver breakthrough customer value. Facebook wasn't going to gradually update itself into Twitter any more than Vista would gradually update itself into Windows 7. You must focus on what the customer is trying to accomplish, and sometimes that isn't a quick change.

Eric Aside The best way to learn how to ship is to do it early and often. Make every build a shippable build. Build every day, and rebuild the entire system at least every week. Deploy regular tech previews and betas. Deploy regular incremental updates and fixes into production. Ship early, ship often. Practice makes perfect.

There are too many of them

However, not everything about shipping packaged products applies to shipping services. There are mental, process, and team adjustments that you need to make.

First and foremost is that services run across hundreds or thousands of machines dispersed in multiple data centers worldwide. Sometimes functionality and data are replicated. Sometimes functionality and data are specialized. Usually, it's a combination of both for scale and reliability. Naturally, this presents design and synchronization problems, but plenty of books have been written about that (read don't rediscover). The less obvious challenges are around debugging and deployment.

Why is debugging a service so tough? Timing issues are killer given multiple threads on multiple processors across multiple machines. Yikes! However, that's not even the toughest challenge.

What's the first thing you do when debugging an issue? Analyze the stack, right? With services the stack is split across servers and requests, making it nearly impossible to trace a specific user action. The good news is that there are new tools that help tie user actions together across machines. The bad news is that this isn't the toughest challenge either. The toughest challenge is that you're always debugging in the live environment. You don't get symbols, breakpoints, or the ability to step through code.

So let's recap. Debugging services means debugging nasty timing issues across multiple machines with no stack, symbols, or breakpoints on live code. There's only one solution—instrumentation—and lots of it, designed in from the beginning, knowing you'll soon be debugging across live machines with no stack, symbols, or breakpoints.

They're multiplying too rapidly!

Solving debugging brings us to the other huge challenge—deployment. Deployment needs to be completely automated and lightning fast. We're talking file copy installation, with fast file copy. No registry, no custom actions, and no manual anything.

Why does deployment need to be so fast and simple? Two reasons:

- You're installing onto hundreds or thousands of machines worldwide while they are live. Installation must work and work fast with zero human intervention ever. The slightest bit of complexity will cause failures. Remember, five minutes times 1,000 machines equals three and a half days. It had better just work.
- The number of servers needs to grow and shrink dynamically based on load. Otherwise, you are wasting hardware, power, cooling, and bandwidth in order to meet the highest demand. Because your scale depends on load, it can change any time. When it changes, you need to build out more systems automatically and instantly.

The happiness around deployment is that Azure will do most of the heavy lifting for you (so let it, don't reinvent). However, you still need to design your services to support file copy installation.

Life is so uncertain

Enough of the challenges you can predict, how about the unpredictable ones? The services landscape is in constant change. While some services are sticky because they hold your data (like Facebook or eBay), many aren't sticky at all (like search or news). A few minutes of downtime can cost you thousands of customers. Data compromise or loss can cost you millions of customers. They'll just switch. Our competitors will be happy to accept them. It cuts both ways, so you need to work hard to both welcome and keep new users.

When you update a service everyone gets the new version instantly, not over years. If there's a bug that only one customer in a thousand experiences, then that bug will hit thousands of customers instantly (law of truly large numbers). That means you need to resolve the issue quickly or roll back. Either way, it's a bad idea to update a service on a Friday and a good idea to have an emergency rollback button always at the ready.

Finally, it's important to realize that services are living, changing things. You'd think that because the servers are all yours, with your image and your configuration, that it would be a controlled environment—and it is until you turn on the switch. Once the server goes live, it changes. The memory usage changes, the data and layout on the disks change, the network traffic changes, and the load on the system changes. Services are like rivers not rocks. You can't ship and forget services. They need constant attention. To make your life easier, bake resilience in by automating the five Rs—retry, restart, reboot, reimage, and replace (though replace may require human hands at some point).

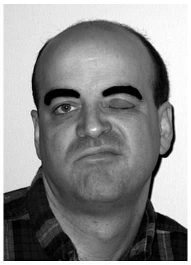
The happiness that comes with these heartaches are customers willing to switch; an ideal idea-testing platform because you can show customers different ideas and see which they prefer on a daily basis; and the ability to ship now and find the tricky intermittent Heisenbugs later (using your five Rs's resilience to keep up availability).

Back to basics

There you have it. Some food for thought mixed in with the old basics of writing solid code that focuses on customers and their goals.

However, none of this is worth anything without shipping. Make shipping a priority and we all win. Sure, the quality bar has gone up, but we're not kids selling lemonade anymore. We need to ship quality experiences regularly, on both long and short time scales. We need to ship on the Internet, on the PC, and on the phone. We need to serve our customers well and delight them into sticking with us. It's a long journey, but it doesn't start until we ship.

September 1, 2009: "Right on schedule"



My older son can now drive. This adds two new worries to my life—how ancient I feel and thoughts of my son in a ditch somewhere. To mitigate the second worry, my wife and I enforce a curfew and insist that my son call if he's running late. The other night, he arrived home 20 minutes late without notice. My wife was furious that he was late. I was furious that he didn't call.

Why didn't my son call to say he was running late? Because, like my wife, he was focused on the schedule. He avoided facing conflict until he got home. He said, "I got home as fast as I could"—presumably breaking numerous traffic regulations along the way. My son completely missed the point. The purpose of the rules was to mitigate risk, yet his response to them was to drive recklessly.

Software engineers do this all the time. They come up with a development schedule, unexpected issues come up, and they end up being late. Instead of informing their managers of the delay, they avoid facing conflict, rush the work, sacrifice quality, and slip the schedule, all with little control or visibility. It's the opposite of what managers should want, yet those same managers insist on following the schedule precisely. Why? Because most managers and engineers don't distinguish between the two types of scheduling—meeting a commitment and managing risk.

Eric Aside I love this particular column. It covers something critical and basic, yet widely misunderstood. I wish my family, my friends, my coworkers, and my community would all read it.

Those who understand binary and those who don't

Yes, that's right. There are two types of scheduling and project management.

- **Meet a commitment.** You made a commitment to customers or partners, and you must meet it at the quality and time period promised. Period.
- **Manage risk.** There is a mix of critical and desirable work. People can make bad choices. Issues can arise. You must manage risk to ensure critical work gets done.

These two approaches to scheduling and project management often get confused. Why?

- **They typically appear together.** The overall project has commitments, but it is made up of smaller tasks that require risk management.
- **They both use dates.** The difference is that dates for commitments are untouchable and drive everything. Dates for risk management are simply checkpoints to make sure work stays on track.
- **They both are called scheduling.** Most people simply don't know the difference.
- **Meeting commitments is the only type most people are taught.** From the time kids enter school, they are exposed to inflexible due dates and commitments they must meet. When they later learn project management, it is all about Gantt charts and milestones, with some risk management thrown in as an aside.
- **Managing risk is usually self-taught and informal.** A small number of people are formally taught risk management. Most of us learn it from peers in college as we juggle large workloads. Instead of completing everything on time, we form workgroups, focus on the critical work, and minimize the damage to our grades.

This tragic lack of understanding leads to horrible decisions, poor engineering, and scheduling disasters. You need to know the difference and apply the right scheduling to the right problems. Let's start with meeting commitments.

That's the only thing you're committed to

Meeting commitments is essential to working with partners, which in turn is essential to running most businesses. You can't coordinate work across internal dependencies or external agreements without synchronizing on dates and deliverables. Because commitments cascade, you must meet them or face catastrophe.

Say your daughter's birthday is coming and you've promised her a new game she wants. How would you feel if it wasn't delivered to you when promised? There is a chain of handoffs between the developer, the manufacturer, the seller, and you that all must be met to ensure your daughter's happiness on her birthday.

Of course, this is much easier to do for web-based products, but the same problems come up with handoffs between teams or departments. Delivering on commitments builds trust and lasting relationships between partners. Missing commitments does the opposite. While many software projects require little or no coordination across groups, running large and mature projects typically involves meeting commitments, and thus formal project management techniques.

Eric Aside To help make their commitments, companies typically use inventory, buffers, and other forms of risk management for individual steps in the process. That's the whole point. You use formal project management for meeting your high-level commitments and risk management for properly completing the individual steps in your process.

Don't you think it's a little risky?

Risk management is about making sure the critical work gets done properly, even in highly variable environments. Scrum and Feature Crews are excellent examples of software development practices that focus on risk management—in particular, the risk that you won't efficiently ship value and quality to the customer.

As I pointed out in my very first column, "Dev schedules, flying pigs, and other fantasies," dev schedules and test schedules are in the risk-management category. All the feature dates are checkpoints to mitigate risk. Only the small number of cross-group synchronization points (major milestones) are commitments.

What matter in risk management are focus, order, and status—not precision. Focus on what's important, do those items in priority order, and track the status as the situation changes. Notice that hitting precise dates for tasks isn't important, so long as you finish the critical tasks at the expected quality on time. Everything else can be cut.

That's why you must tell your engineers the same thing I told my son: "Coming home right on time isn't important. Telling us you aren't coming home on time is." You can only manage the risks if you know about them. The dates are there only to alert you when plans need to change.

Of course, you can't slip a critical task past a commitment date (a major milestone), and my son can't stay out past 1:00 A.M. or his license will be suspended. But curfew is well before then, as it should be, to avoid any chance of catastrophe.

Eric Aside There are many popular risk-management techniques. Here are a few handy ones for software development (many taken from prior columns):

- ❑ Hold daily stand-up meetings—15-minute meetings to talk about progress, future work, and blocking issues (called scrums in Scrum).
- ❑ Assign a backup for all task assignments (partner less experienced folks with more experienced folks).
- ❑ Use buffers—set aside time for task fluctuations (personally, I never liked this practice; I'd rather have a well-prioritized list with no intention of completing everything).
- ❑ Under promise and over deliver—also known as setting appropriate expectations.
- ❑ Establish a fallback plan—have a plan in mind in case a risky task fails, like cutting back the feature or going back to the previous version.
- ❑ Balance risk—keep the overall risk of your project at a constant state of "scary but not terrifying" by adding and removing risk as things change. For example, if a team member has a parent fall ill, your risk has increased, so you should cut a risky feature or reassign that tough task you gave a junior engineer.

You pick the one right tool

So before you start working on a schedule, stop and think about what's on it. Is it a set of dates and deliverables you've committed to a partner? Or is it a set of tasks with various priorities that you need to track and avoid messing up?

In the case of my son coming home by curfew, it was a task we didn't want him to wreck, literally. Thus, we were doing risk management and the focus needed to be on giving timely status as opposed to coming home at a precise time. Most software development tasks fit that mold. You just want your engineers to tell you promptly if they are running late so you can adjust. Precision is unnecessary and potentially counterproductive.

However, if you are running a large project with multiple teams and partnerships, then commitments and synchronization are critical at a high level. The high-level schedule is full of milestones and classic project management tools.

Just don't confuse the high-level schedule with the low-level tasks. If you treat the low-level tasks like your high-level commitments, your engineers will take shortcuts and drive too fast. Instead of managing risk, you might cause them to crash one of your critical tasks, which in turn breaks your high-level commitments. Use the right tool for the right level. You'll sleep better at night.

Eric Aside For more on combining traditional project management techniques with agile project management techniques, read the next column, "Coordinated agility."

May 1, 2010: "Coordinated agility"

I've been using Scrum for seven years and writing about it for the last six. Scrum's concept is fantastic—multidiscipline, self-directed teams iterating on short scenarios (stories) in small batches from start to finish, within short, fixed-length, continuous-improvement cycles. Given the success many Microsoft teams have had with Scrum, it's stunning that such a strong disconnect still exists between org-level project managers and team-level Scrum engineers.

Many org-level project managers and middle managers believe Scrum is chaotic, haphazard, dangerous nonsense that discourages planning. Many Scrum aficionados believe project planning is wasteful, disruptive, unnecessary horse manure that serves only to placate out-of-touch upper management with fantastical schedules. Well guess what? You are both wrong, and far too smart to be so stupid as to assume the other is ignorant. And yet, you are both so very ignorant.

Scrum is loaded with planning, and it efficiently tracks more data in more detail than any other project management method I've seen at Microsoft (except for TSP, used by a few teams). Likewise, high-level project planning is critical to successfully scoping, coordinating, and delivering any large-scale initiative. If you have limited vision, then Scrum alone is fine. If you want to deliver lower quality and less customer value in more time than your competitors, or if you want to micromanage every aspect of your limited scope, then project planning alone is fine. If you have a bold vision with broad scope that you want delivered efficiently with high quality and copious customer value, then you need a balance of both high-level project planning and Scrum.

Eric Aside Feature Crews is an interesting variant of Scrum. This practice originated in Microsoft Office. Like Scrum teams, feature crews are multidiscipline, self-directed teams iterating on short scenarios (features) in small batches from start to finish. While they may have daily stand-up meetings, feature crews don't typically follow the Scrum model of fixed-length sprints and highly iterative planning. Nonetheless, feature crews have become a staple of many Microsoft teams and effectively implement a lean software engineering process.

I respect your right to disagree with me

Why is there a disconnect between many org-level project managers and Scrum aficionados? I talked about the cause a few years ago in my introduction to this chapter on project mismanagement:

> *Project management happens differently at different levels of scale and abstraction. There is the team or feature level (around 10 people), the project level*

(between 50 and 5,000 people working on a specific release), and the product level (multiple releases led by executives). Agile methods work beautifully at the team level; formal methods work beautifully at the project level; and long-term strategic planning methods work beautifully at the product level. However, people rarely work at multiple levels at once; in fact, years typically separate those experiences for individuals. So people think effective methods at one level should be applied to others, which is how tragedies are often born. The moral is: small tight groups work differently than large disjointed organizations. Choose your methods accordingly.

You can’t expect process-heavy, formal methods to work well for small teams, any more than you can expect dynamic, emergent planning to work well for large organizations. To bridge the gap between the two, you must understand the goals of each and what they need from each other. Let’s break it down.

Plans are nothing; planning is everything

The goals of high-level project planning (vision, architecture, schedule, and risk management) are to:

- **Set a compelling shared vision that aligns a large organization.** Without a compelling shared vision, your long-term success is left to chance. Sure, iterating with customers can provide greatly appreciated incremental value, and even hint at long-term value. But major advances are born of compelling shared visions.
- **Establish interfaces between teams that enable the vision.** A compelling vision gives an organization a shared destination. Establishing interfaces between teams and components provides an architectural roadmap to that destination. Sure, you could drive from Seattle to New York without a map or highways—but it would take you a long time to get there.
- **Meet commitments to business partners.** Big, established, successful companies get that way through partnerships. If you are delivering a compelling vision with real value, you’ll want to involve your partners. That means commitments on both sides. Real money is on the line.
- **Reveal and resolve issues that could jeopardize those commitments.** Any large project with partnerships involves dependencies, risks, and coordination. There are numerous ways the project could stall or fail, including greater than anticipated success after launch. You must plan for success and failure. It’s hard work to reveal and resolve issues before they sabotage a project.

Project planners, architects, and middle managers need Scrum teams to align to the shared vision, abide by their interfaces (or update them collaboratively), meet their commitments (or update them collaboratively), and surface and mitigate issues as they arise.

I can take care of myself

Okay, we have a shared vision, interfaces, commitments, and a risk-mitigation plan. Now we just need to follow them, right? What universe do you come from? In my universe, change is the only constant, and progress happens one step at a time.

At a high level of abstraction, the plan might come together and make perfect sense, but at a low level, details intercede. Variation and complexity surface as the details emerge. Project planners, architects, and middle managers could try to micromanage these changes themselves through endless status and design meetings, adding tons of overhead, creating bottlenecks, and grinding progress to a crawl. Or maybe they could trust the engineers closest to the details to work through the problems.

Agile methods like Scrum adjust to emergent details and changing situations quickly and effectively. They minimize work in progress and overhead in favor of working, customer-focused solutions that meet project requirements. That doesn't mean Scrum teams don't plan—it means they approach planning in an iterative manner as they take each step toward the shared vision.

Scrum teams need project planners to set the vision, interfaces, and prioritized requirements and then get out of the way. So long as the vision is reached, the interfaces respected, and the requirements met, project planners should be very happy. That self-directed Scrum teams quickly resolve most of the issues themselves is something project planners and middle managers must learn, accept, and in time, embrace.

Eric Aside Many project planners and middle managers get nervous at the idea of trusting self-directed teams. There are three effective ways of alleviating those fears:

1. Keep track of the data—Scrum surfaces loads of data around estimates, priorities, work in progress, blocking issues, velocities, and completed work.
2. Have weekly or daily Scrum of Scrum meetings—15-minute meetings with all the Scrum Masters to discuss blocking issues and review progress.
3. Set a ground rule for Scrum team decision making: any decision that impacts more than two Scrum teams must be reviewed by the project team or architecture team.

So happy together

Project planners and Scrum engineers should embrace each other. (Group hug!) They complement each other's work. Project planners give Scrum teams direction. Scrum teams allow project planners to focus on the broad picture, while the Scrum teams provide detailed information on their rapid, flexible, iterative progress toward high-quality, functioning, customer scenarios.

Both project planners and Scrum engineers play critical roles on a project with a bold vision and broad scope. Ignorance is no excuse for fear or rejection of each other's work. The better we understand each other's roles and goals, the better we are able to deliver delightful and innovative experiences to our customers.

Eric Aside What happens when you don't balance project planning and self-directed teams? At a superficial level, Apple and Google provide interesting examples.

Apple is all about project planning from the top, with micromanagement down to the bottom. The result has been bold vision for their products, but limited breadth for their business.

Google is all about self-directed teams with minimal management oversight. The result has been a broad range of efficiently produced services, but those services are disjointed due to a lack of shared vision beyond the company's general mission.

This is an overly simplistic view to be sure, but an enlightening one nonetheless. Of course, Apple and Google have been very successful, even with their limitations. Balancing solid project planning with efficient self-directed teams allows Microsoft to better compete with both.

Chapter 2

Process Improvement, Sans Magic

In this chapter:

I'm wrong, okay? I know nothing. Now calm yourself! Some people raise process dogma to the level of religious fanaticism. My own pet theory about why relates to superstition. B. F. Skinner noted that superstition arises when animals, like pigeons, associate chance behaviors with desired results. People get locked into very particular practices when by a combination of chance and skill they achieve great outcomes.

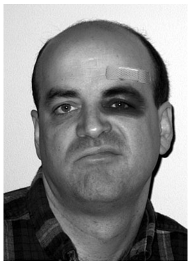
Not that there's anything wrong with that, I enjoy superstitious behavior as much as anyone. But when people become inflexible in their application of methods, say eXtreme Programming, superstition becomes counterproductive at best, divisive at worst.

In this chapter, I. M. Wright analyzes a wide collection of process improvements and techniques, minus the superstition. The first column was part of an Interface *issue focused on Six Sigma at Microsoft (many of the original "Hard Code" column topics were set by the* Interface *editorial staff). This is followed by columns on Lean software engineering, traceability of requirements, practical application of Agile techniques, defining metrics that won't be gamed, managing dependency relationships, good bug management, continuous deployment of services, and reducing cycle time.*

Excellent books have been written about many of these topics in far more depth than I provide here. If I. M.'s presentation of these concepts doesn't match your precise ideals, please forgive him. After all, he's not trying to be perfect, just right.

—Eric

September 2, 2002: "Six Sigma? Oh please!"



I'm sorry. If you talk to me about yet another totally continuous quality management improvement program, I might have a seizure. Now we're experimenting with the Six Sigma problem-solving methodology.

In only five days over eight weeks, you can be trained as a Six Sigma Green Belt. Or go for it all—in just four months become a Six Sigma Black Belt. I think I'm going to hurl.

I just don't understand why we need buzz words and "Karate Kid" references to apply good engineering practices to our problems. It's like senior managers leave their brains, education, and experience at the door and get seduced into thinking that the latest fashionable regurgitated metric analysis fluff will solve all the ills of our unenlightened workforce.

Eric Aside I'm constantly confronting management in my columns. Along with PMs, managers are one of I. M. Wright's favorite targets for ridicule. To their enormous credit, Microsoft managers have never taken it personally, and many are avid fans. Sure, my manager has occasionally been asked if these columns are sufficiently constructive. But in 10 years of writing on fairly contentious topics, I've never had a column censored or altered by management.

But I work at Microsoft under these managers, so I had to read the articles in this issue of *Interface* focusing on Six Sigma and the material on the Six Sigma website, like it or not. Am I blown off my feet? Please. Is the content filled with new and exciting ideas that will revolutionize the way we produce our products? As if. Is there anything there of merit? Of course.

Egads! What sorcery is this?!

Six Sigma is a structured problem-solving system with a "toolbox" of techniques used to analyze and interpret issues for all kinds of business, development, and manufacturing processes. The actual techniques themselves are nothing new—brainstorming, the five whys, cause and effect diagramming, statistical analysis, and so on. These techniques have been used for years to discover the root cause of issues in engineering and business.

The methodology is based on tried and true problem-solving principles that date way back: define, measure, analyze, improve, control. This basic cyclic approach to quality improvement is used in just about every product group at Microsoft during stabilization. Bugs are defined (spec'd), measured (found and documented), analyzed (triaged), improved (fixed), and controlled (regressed, prioritized, and triaged again).

So why have a Six Sigma group? Why become a Green Belt? What's with having 20 full-time Six Sigma Black Belts at the company?

Calling in the cavalry

Basically, in the heat of the moment we panic and forget all the engineering knowledge and practices we have learned and know so well. That's everything we knew before the pressure crushed us or we became so engulfed in the problem that we no longer could see the dead tree for all the bugs.

So you call your local Green Belt, or bring in the big old Black Belt, and he reminds you of what you should have been doing in the first place. However, because of the highly structured nature of the Six Sigma system, all the passion and personalities get removed.

Instead of placing blame or getting caught up in guesswork and blind alleys, the Six Sigma folks look dispassionately at the real data and derive what's actually wrong and what can be done to improve the problem. Then they leave you with a process to track your improvement and control its effects.

Creating order out of chaos

Yes, anyone with a good engineering background could find the same problems and fix them. Anyone who made it through interviews at Microsoft should have the intellectual horsepower to figure out a solution to a problem. But sometimes when you're in it too deep and tempers are flaring, you need an outside calm influence to help you get centered and focused on doing the right things.

In addition, the Six Sigma folks get exposed to a wide range of techniques and best practices from around the company. They can bring those experiences to your group and come up with interesting solutions that may have escaped your notice.

Does this make me a Six Sigma booster? Nah, I still think the idea of Green Belts and Black Belts is goofy and that the methodology itself is recycled TQM and CQI. However, Six Sigma is the process that Microsoft has chosen to experiment with—if there's a group that can come in and help when problems get out of hand, that's a good thing to me.

Eric Aside While Six Sigma never quite took hold in product development at Microsoft, the concept of having coaches and groups you can turn to in a pinch did. I used to be the manager in just such a group.

October 1, 2004: "Lean: More than good pastrami"

Ever walk through a public space, like an airport terminal or public park, and get accosted by crazies trying to convert you or scare you or assault your supposed ignorance? Get into a conversation with one of these people and logic and reasoning become ludicrous. Everything to them is blind faith and irrefutable truth. Even if you wholly agree with them, there is still no room for questions or analysis. You must believe, you cannot question, even in part.

This makes me sick. I mean truly physically sick. I was given a mind of my own, and I fully intend to use it. Not just at parties and social occasions but on every subject and dealing I have. Questioning why and understanding how are at the center of who I am.

You'd think my sensibilities would be the norm for software developers, who can't debug what they don't understand. But the same zeal that some folks devote to religion, political battles, and environmental concerns also gets directed by some developers toward new development practices like eXtreme Programming (XP), Agile, and the Team Software Process (TSP).

All things in moderation

I love many of the ideas and approaches advocated by these development paradigms. But if I question a true believer why a certain thing is done or suggest a small change in a rule or practice to better adapt it to my work, look out! It's like showing a ring of power to an old hobbit—the fangs come out, the hair raises on end. For some developers, eXtreme Programming and the Agile Manifesto have become a cult. For some developers, TSP is a measure of allegiance—you're either with us or against us.

Well excuse me for being practical. Excuse me for using my head. Excuse me for doing something because it's useful instead of magic. I don't do things because "That's the way you must do it." I do things because there's a darn good reason why they work, and there are also good reasons why working some other way fails.

Eric Aside There, I feel better. Often these rants that lead columns overstate my own feelings on a subject, but not this time. There's harmless superstition and then there's lunacy. I'm not a big fan of lunatics.

Waste not, want not

Which brings me to Lean. Ah yes, the title of the column. While there are many wonderful things in XP, Agile, and TSP, there is at least one concept that they all have in common:

reduce wasted effort. That is the focus of Lean Design and Manufacturing, a concept from Toyota that predates XP, Agile, and TSP by more than 30 years. While XP, Agile, and TSP attack the problem of waste in different ways, we can better understand what each is doing by using the Lean model.

So, at the risk of offending some zealots’ sensibilities, let’s break it down. Lean focuses on delivering as much value as possible to the customer with a minimum of wasted effort. It accomplishes this by using a pull model and adopting continuous improvement. The pull model means simply, “Don’t do work until it’s needed.” This reduces unused, unnecessary, and undesirable work. The continuous improvement is focused on reducing waste and creating a smooth-flowing stream of customer value.

Eric Aside Kudos to Corey Ladas who first introduced me to Lean, as well as Axiomatic Design, Scrum, Quality Function Deployment (QFD), Set-Based Design, Kaizen, Pugh Concept Selection, and who knows how many other great ideas. We worked together for two productive years, and he’s left my team with a hole that cannot be easily filled. He’s got a Lean Software Engineering website now with another great former team member, Bernie Thompson.

Lean defines seven types of waste that disrupt the flow of customer value:

- Overproduction
- Transportation
- Motion
- Waiting
- Overprocessing
- Inventory
- Defects

These are obviously manufacturing terms, right? They can’t possibly be relevant to software, right? Oh, to be young and foolish. All seven of these sources of waste are directly related to software development. I’ll treat them like the seven deadly sins and talk about how XP, Agile, TSP, and plain common sense can help.

Overproduction

The first deadly waste is producing more than you need. Like this never happens. Has a product ever shipped without cutting features that were already spec’d and coded? Has a product ever shipped without keeping features customers never use? Too complex, too general, too extensible, too fancy, too redundant, too convoluted. Overproduction is a killer. It’s an unbelievable waste.

XP solves this with short and tight iterations. It insists on constant contact with customers and constant communication between developers. This ensures that everyone knows what others are doing and the customer always thinks it's a good idea. As a result, almost all the work that gets done is of value to the customer. Of course, the Microsoft customer is super-sized, so many Microsoft teams have turned to Agile.

Agile is a collection of Lean practices, including XP. Because Agile is more of an alliance than a specific technique, it provides a number of interesting approaches to development. One of these is a project management practice called *Scrum* (named after the rugby term). Teams meet with the customer's representative regularly, usually every 30 days, to demonstrate progress, reprioritize items, and make process improvements. As with XP, team members also meet daily to keep tabs on each other's progress and any blocking issues.

By reprioritizing work monthly and reorganizing work daily, a Scrum team tunes itself to only what's important to the customer. Little work is wasted. By focusing on process improvements at regular intervals, the value stream can be constantly optimized.

Go deep

Of course, you can use Scrum and XP poorly by making the customer wait for value while you work on "infrastructure." There is a fundamental premise behind quick iterations built around regular customer feedback: develop the code depth first, not breadth first.

Breadth first in the extreme means spec every feature, then design every feature, then code every feature, and then test every feature. *Depth first* in the extreme means spec, design, code, and test one feature completely, and then when you are done move on to the next feature. Naturally, neither extreme is good, but depth first is far better. For most teams, you want to do a high-level breadth design and then quickly switch into depth-first, low-level design and implementation.

This is just what Microsoft Office is doing with feature crews. First, teams plan what features they need and how the features go together. Then folks break up into small multidiscipline teams that focus on a single spec at a time, from start to finish. The result is a much faster delivery of fully implemented and stable value to demonstrate for customers.

Eric Aside Naturally, the idea of feature crews isn't new. However, finding a way to implement Lean software development within a huge live production environment like Office is a major achievement. Keep in mind, Office is now a system of multiple desktop applications, server applications, and online services.

Depth first reduces overproduction by staying focused on work that can be used, rather than on "infrastructure" that may never be leveraged or a little bit of everything that may never stabilize. Another great method for depth-first development is Test-Driven Development (TDD), but I'll save that for the overprocessing section.

Transportation

The second deadly waste is waiting for stuff to arrive. In manufacturing, this typically means the transportation of parts. For software, it's the transportation of deliverables between teams. There are three nasty sources of transportation issues: builds, branches, and e-mail.

- **Builds** The longer the build, the bigger the waste of time. Like I need to tell you this. XP and Agile both insist on daily builds, a rule they may well have gotten from Microsoft. For huge teams, a daily build has become a fantasy. Luckily, we have good people working on the issue, but it's a big problem. Enough said.
- **Branches** I love Source Depot. It's been huge for the company. But the frigging thing has become a pet elephant. Sure they're cute when they're a baby, but in a few years you're constantly either feeding or shoveling, and your mobility suffers. While branching is great, many large teams have taken to branching branches. So if you are on branch A2.B3.C1 and your buddy with a key feature or fix is on branch A3.B1.C2, your buddy needs to reverse integrate C2 into B1 then B1 into A3, and then you have to integrate A3 into A2 then B3 then C1. AHHHHHHHHH!!!!!!!! You might as well watch grass grow. The solution is one level of branching off your current release line, period.

  Eric Aside Source Depot is the large-scale source control system Microsoft uses to manage hundreds of millions of lines of source code and tools, including version control and branching.

- **E-mail** The last transportation nightmare is e-mail notification: PM telling dev and test that specs are ready; dev telling test that code is ready; test telling dev it's blocked on a bug; dev telling PM it's blocked on a design change; and, my personal favorite, any kind of communication between a client and dependency or vendor, particularly overseas. XP and Agile solve the e-mail notification problem by removing the roles and having the team meet daily. For remote vendors and dependencies, this can't work. For now, we must rely on automated notification where possible, Live Meeting where reasonable, and clear e-mail that answers anticipated responses to reduce roundtrips everywhere else.

Motion

The third deadly waste is spending time just finding stuff. On the manufacturing floor, it's the wasted motions of robots and people. In the software world, it's time spent figuring out what to do, where to go, and how to fix. Poor search technology is a great example of wasted motion. So is untestable, unmaintainable, unmanageable code.

Using asserts and validating input help find bugs faster and reduce wasted motion. So do design reviews, code reviews, code analysis, and unit testing. XP even suggests pair programming, but personally, I think that wastes resources (except for devs learning a new code base). TSP measures all your activities and defects, which allows you to study exactly how your time is spent and significantly cut down on your wasted motion.

Eric Aside My team has since adopted pairing for creating new content in unfamiliar areas. It works extremely well.

One particularly annoying and avoidable source of wasted motion is duplicating bug fix information for code comments, Source Depot, Product Studio, and check-in mail. And everyone wastes motion managing multiple copies of bugs and project schedule data. Tools that make these things easier by entering the information once and automatically populating it to all other places can go a long way toward reducing deadly motion sickness.

Waiting

The fourth deadly waste is waiting around for work. Transportation issues cover a big part of waiting for builds, branch integrations, and timely communication. But there are plenty more places to wait. The most common dead zone is caused by teams not agreeing on the priority order of features or simply not following the predetermined order. That is, PMs writing specs out of order so devs have to wait. Devs writing features out of order so testers have to wait. Testers writing tests out of order so everyone has to wait.

XP, Agile, and TSP all force teams to decide on a priority order, get buy-off from the customer or their representative, then work in that order until they decide to review the priorities again. TSP is particularly rigorous in this way, but also can be less iterative about plans without a flexible leader.

Another source of waiting is unstable code. As long as the code is unstable, the test team has to wait, as do any other mechanisms you have for customer feedback. XP and Agile put a premium on verifiably stable code, another essential element of the depth-first strategy.

Eric Aside Another form of waiting is for service environments to stabilize or unfreeze as the result of new service deployments or other special events. The best way to avoid this wait time is exposure control and continuous deployment, which I discuss later this chapter in "There's no place like production."

Overprocessing

The fifth deadly waste is over-engineering. You see this all the time in the form of producing overly complex features, fine-tuning performance in areas that already perform adequately or aren't the true bottleneck, and adding generalization or extensibility when it isn't required. This waste is related to overproduction but focused on specific feature implementations.

The cure: Test-Driven Development (TDD). TDD is an XP and Agile technique for implementation design. As a side benefit, it provides unit tests with full code coverage. The process is fairly simple:

1. Define your API or public class methods.

   Eric Aside This is a point of contention between me and some members of the Agile community: do you define your API or public class methods before writing unit tests or after? Purists will say after; I say before. The difference is the amount of up-front design and the nature of your relationship with outside groups that depend on your code. I tackle up-front design in other columns, which when taken in moderation, I believe is essential to success in projects with more than 100,000 lines of code.

2. Write a unit test for a requirement of the API or class.
3. Compile and build your program, and then run the unit test and ensure that it fails. (If it passes, skip step 4.)
4. Write just enough code to make the unit test pass. (Also ensure that all previous unit tests continue to pass.)
5. Repeat steps 2 through 4 until all API or class requirements are tested.

Naturally, after you get the hang of it, you can write unit tests for more than one requirement at a time; but when you first get started, try doing just one. It builds the right habits.

When you use TDD, you don't write any more code than is absolutely required. You also automatically get easily testable code, which usually correlates to strong cohesion, loose coupling, and less redundancy—all very good things indeed. Oh, and did I mention you also get unit tests with full code coverage? What's not to like?

Inventory

The sixth deadly waste is undelivered work product. This is related to cut features, but it also includes the amount of work in progress. When you develop breadth first, all your work is in progress until the code is complete and stable. All the completed specs, designs, and code that are waiting to pass tests are inventory. Their value is not yet realized.

Unrealized value is wasteful because you can’t demonstrate the value to customers and partners. You can’t get their feedback. You can’t improve and optimize your customer value stream. Of course, if product plans change, this unrealized inventory often becomes a huge wasted effort.

The Lean pull model of working only on things as they are needed drives low inventory, as demonstrated in Scrum and TDD. Scrum pays special attention to work in progress, tracking it and working hard to minimize it. Scrum also leverages regular opportunities to improve and optimize the way you deliver value. TDD has you implement code only as needed to satisfy requirements, and no more.

Defects

The seventh deadly waste is rework. It’s the most obvious one and the one I’ve ranted about incessantly in the past (see Chapter 5, “Software Quality—More Than a Dream”). XP and Agile get at reducing bugs and rework by a variety of techniques, not the least of which is TDD, daily builds, continuous code reviews, and design reviews.

However, XP and Agile also reduce bugs in a more subtle way—by creating a structure where you learn as you go. Using depth-first development, you figure out parts of the project step by step before you’ve designed and coded the whole product. This prevents serious architectural issues from remaining concealed until it’s too late to adjust. Sound familiar?

Reducing defects is a specialty of TSP. Teams using TSP have dropped their bug rates by a factor of a thousand from the industry average. I wrote in detail about the TSP defect prediction, tracking, and removal approach in “A software odyssey—From craft to engineering” in Chapter 5. While TSP isn’t inherently lean, it doesn’t preclude depth-first development either.

Symbiosis

This brings me to the point where I get to infuriate the XP, Agile, and TSP true-believers. There’s no reason why you can’t combine these techniques to be greater than the sum of their parts. Use Scrum to drive a lean, depth-first, flexible, and optimized development schedule. Use TDD to create a lean implementation. And use TSP to analyze your defects and your work, which will result in vastly reduced bugs and wasted effort. While that may be heresy to some, it sounds like common sense to me.

Now if I could just find some good pastrami.

Eric Aside I grew up in New York. It’s tough to find good pastrami in Redmond.

April 1, 2005: “Customer dissatisfaction”

You always hurt the one you love. We must really love our customers. We ship buggy code, though that’s not the big problem. We miss our ship dates—not the big problem. We don’t have a clear, broad, and prioritized understanding of customer needs, but that’s not the big problem. We don’t communicate to our customers well and with one voice, but that’s not the big problem. We don’t listen to our customers well and then transfer that information to the right people; but again, that’s not the big problem. No, the big problem is that all too frequently we have no idea when we are tormenting our customers and how badly we’re doing it until it’s too late.

Eric Aside I am severely overstating the case here for dramatic effect. In fact, we do a very good job of listening to our customers and integrating that value into our products. It’s been a huge competitive edge for Microsoft over the years. Regardless, our customers’ expectations have risen as the software market has matured, so to keep our competitive edge we must continue improving. This column discusses the advantages of tracing every code change back to the customers who needed or requested the change.

If we ship bug-free, high-quality code, but it wasn’t what the customer wanted, then customers are dissatisfied. The same is true for shipping undesirable code on time. Even if we do have a clear, broad, and prioritized understanding of customer needs, we still have to ship code that meets those needs or the customer will be dissatisfied. Communicating well and listening well aren’t enough. Nothing counts if we don’t deliver what the customer wanted.

Ignorance is bliss

In fact, good communication and listening actually hurt us. Say we talk to some customers and find out exactly what they want. The customers are pleased that we listened to them, and they know that we know just what they need. Two years later, we deliver a solution that falls short of their expectations. Uh oh. Now the customer is

- Disappointed because the product doesn’t perform as desired.
- Insulted because we wasted their time and raised, then dashed, their hopes.
- Incensed because we broke our commitment to serve them. They may never trust us again.

At least if we had ignored the customer, we could have excused the mistake. “We’re ignorant ninnies” could be our claim. Unfortunately, we did know, we did acknowledge, and we did commit. Even if the commitment wasn’t legal and contractual, it’s a commitment nonetheless, and we broke it.

Too much, too late

Think this doesn't happen? You are so wrong. We break our commitments all the time. It's amazing that we still have customers. Our salespeople talk to customers and tell them about our plans. Our consultants visit customers and say they'll work with product teams to enable certain solutions. Our marketing and product planning people run focus groups and tell customers, "We're working on it."

Eric Aside When I say, "We break our commitments all the time," I mean we fall short of the ideal. We deliver what the customer requested but not what they really wanted or needed. Customers don't know what they want till they see it. That's why many Agile methods focus on iterative customer feedback. I use the word "commitment" because it has strong connotations for Microsoft employees. It is too easy to let "little" problems persist in our products, yet it's those little problems that cause big headaches for customers and I want engineers to feel that.

And we do work on it. Market opportunities drive our product plans and visions. But we don't close the loop with customers until it's too late. Heaven forbid that when a customer clicks a button in a working product, we should have time to rethink what we've done. Heck, most of the time we've already gotten past code complete before customers touch the product.

Eric Aside Let me promote betas and technical previews here, since I failed to mention them in the original column (a significant oversight). Most Microsoft products use betas, but only one or two and often late in the development cycle. However, a number of products are starting to use technical previews and betas early and often—a practice I adore.

In fact, we're getting really good at closing the loop with customers *after* we ship. Watson and SQM tell us all about the horrible experiences our customers are having with our shipped products. It's a phenomenal step forward. We fix the bugs and ship again three months or three years later, and Watson can show us if the annoying problems are gone.

Eric Aside *Watson* is the internal name for the functionality behind the Send Error Report dialog box you see when an application running on Microsoft Windows crashes. (Always send it; we truly pay attention.) *SQM* is the internal name for the technology behind customer experience improvement programs for MSN, Office, Windows Vista, and other products, which anonymously aggregate customer usage patterns and experiences. (Please join when you install our software; it lets us know what works and what doesn't.)

But what about the problems that cause us to break our commitments, throttle our business opportunities, and shred what little trust we still have with our customers? How do we detect those before we ship? How do we prevent those from happening?

Agile delusions

By now the Agile fanatics out there are screaming, “Use Agile methods!” Yeah, well try meeting weekly or monthly with 100 million customers. It’s not as easy as it looks. I’m not saying it’s a bad concept, I’m saying you’re hallucinating.

Sure, you can have the PM or product planner stand in for the 100 million customers, but the chances of them representing all those customers accurately are similar to you winning the lottery. It happens, a lot of people play that game, but you don’t want to build your business or your retirement on those odds.

You need a direct connection back to the customer that closes the loop, like the connection we have with Watson. Any code you write should map back to a specific customer request, market opportunity, business need (like TwC), or customer issue (like a Watson bucket). That way, if a specific question comes up or you want regular feedback on your progress, you know who of the 100 million customers to call.

Eric Aside In fact, we don’t have a direct connection back to the customer with Watson—the information we get is anonymous and aggregated. What we do have is a direct connection to the customer’s problem. Each “Watson bucket” represents and stores a customer issue that thousands, sometimes millions of customers have experienced. So we don’t know who to “call” with a Watson issue, but we can figure out what their problem was. Trustworthy Computing (TwC)—the Microsoft initiative on security, privacy, reliability, and sound business practices—has produced tremendous gains for our customers from Watson data.

Retracing your steps

So how can you tie a specific customer request to a line of code? For bugs, we’ve come close to doing this, but what about feature development? To figure this out, you must retrace your steps:

- Why are you writing that code? What was the requirement or feature?
- Where did that requirement or feature come from? What was the customer scenario?
- Where did that customer scenario come from? What was the market opportunity or customer engagement?
- Who wrote that market opportunity or ran that customer engagement? What is her e-mail alias?

If you can’t trace the work back to a customer, then you’re hopelessly caught up in guesswork about what the customer really wanted. Traceability is the key to any hope of satisfying our customers.

There's more where that came from

But that's just the beginning. Like all great pivot points, traceability resolves far more than the immediate need of a relevant customer contact:

- Traceability allows our customers to check on the status of their issues and solutions. Customers can do this somewhat today when they check the status of an error or crash because we now have backward traceability, from servicing to product development. Forward traceability, from product definition to product development, is rewarding for us and for customers.
- Traceability helps us prioritize and make tradeoffs, as well as get the features right. Because traceability can connect us with the business impact of our changes, we can intelligently decide the appropriate number of resources to apply to a feature or change.
- Traceability helps us architect solutions, determine dependencies, and organize projects. This is the most unexpected advantage to me. With traceability, you can know what customer scenarios drove what feature development. So you can know how features are related. This determines the right architecture and dependencies, and with them, the appropriate way to organize the project. Amazing.

Of course, without traceability, the customers have no idea if their needs will be met and when; the product group has no idea what the real business impact will be, and therefore can only guess at tradeoffs; and each group has no idea why they need one feature or another and how they depend on each other. So our lives become an intertwined chaotic catastrophe.

Eric Aside Again, I'm overstating the problem for effect, but I'm not overstating the benefits of traceability. I admit it openly. I'm in love with traceability.

The right tool for the job

So how do you get traceability? Ideally, we'd have a tool that traces scenarios and requirements the same way we track bugs and crashes:

- Salespeople and consultants could use the tool to document customer requirements, scenarios, and commitments.
- Marketing people and product planners could use the tool to submit market opportunities, link them to customer engagements, and define key cross-product scenarios.
- Product planners and PMs could use the tool to consolidate requirements; track duplicates; link related scenarios across products; and draft product-level scenarios, requirements, and feature specs.

- Product groups could use the tool to triage feature requests and track their progress through design and implementation.
- Test teams could relate test cases and bugs to scenarios so that the team could easily see the impact of issues.
- The originators of customer requirements could track progress on their requests, be contacted by product teams to clarify issues or contribute feedback, or contact product teams to update requirements when situations change.

Duct tape and baling wire

Some groups are actually trying to use Product Studio for traceability, but it's not the full solution yet. Until we have the right tools in place, there are still ways to trace customer requirements and scenarios throughout your entire design:

Eric Aside Product Studio is our internal work-item tracking database. We productized it as part of Microsoft Visual Studio Team System. In the five years since this column was published, most Microsoft divisions have moved to Team Foundation Server (TFS) to track their projects. We are finally close to realizing traceability as I've outlined it here.

- When you write a market opportunity document, link to the related customer engagement documents and ensure those documents have contact information. Include your own contact information in the market opportunity document as well.
- When you create a high-level scenario, link to the related market opportunities and customer engagements, and again include your contact information.
- When you write a feature spec, requirements list, or product scenario, link to the related high-level scenarios, requirements, market opportunities, and customer engagements. Be specific about which documents relate to which features—don't just create a laundry list.
- When you create design documents, link to the specs and other supporting documents. Again, don't create one long list of references—link to specific information as much as possible. You may even want to pull out specific contacts.
- When you are making tradeoffs or reviewing work, trace all the way back through the links to the people who know. Contact the true source.

Customer satisfaction

Today, we run our business like a child's game of "telephone." Each person tells the next what he thought the customer said she wanted. Each step along the way twists and distorts the message. By the time we ship, the customer doesn't even recognize what she asked for. (This may remind you of the classic cartoon in which the customer wants a tire swing hung from a tree, but instead gets a tree on stilts with a hole in the middle.)

As the product changes and develops, you need to check back with customers to ensure that you're making the right decisions. Equally important is the ability for customers to check with you whenever their requirements or scenarios change. Without a closed loop, this just isn't possible.

Traceability closes the loop, but you have to be aware of what to do and remain diligent. Any breakdown along the way risks breaking a commitment. But getting it right means getting the customer just what he needed every time. And that is a reward well worth the effort.

March 1, 2006: "The Agile bullet"

I'm having a tough time with a decision; maybe you can help. I can't quite decide who is more nauseating: people who use "Agile" methods and wonder why Microsoft can't adopt Agile across the company, solving every ill we face; or people who think the Agile fad amounts to retreaded foolishness preached by ignorant academics to free developers from any sense of responsibility. It's a toss-up; I get the same squeamish feelings in my gut listening to either of them.

Eric Aside This is one of my favorite columns because of the overwhelming love-hate reaction it evoked, often from the same person. Though imperfect, it's a fairly balanced overview of the topic.

Let's get two things straight right now:

- If you think Agile methods fix all that is wrong with how we build products, you are, in fact, a fool. Employing thousands of people to build highly complex and deeply integrated software that hundreds of millions of customers depend on is hard. No one in the world, including those clever folks in the Agile Alliance, knows as much about the task as we do. Not everything we are doing is wrong, and not everything Agile professes is right for our needs.
- If you are an anti-Agile curmudgeon who thinks Scrum is an acronym for a System of Clueless Reckless Untested Methods, you are as much a fool and just as ignorant. Dismissing anything thoughtlessly for whatever reason is prejudicial and unprofessional. Grassroots movements, like Agile, are always grounded in some fundamental truths that can be used to benefit our teams and customers. Those notions may not always fit our business directly, but fundamental truths have a way of applying themselves to any situation when you stop to understand them.

Eric Aside Agile has really been a grassroots effort at Microsoft, led by a wide collection of individuals and small teams throughout the company.

It's time to expel the myths around Agile methods and explain how to use the innovative thinking behind these methods to our advantage.

Enemy of the truth

First, let's break down the Agile myths…

- **Myth #1: Agile = eXtreme Programming (pair programming, Scrum, Test-Driven Development, user stories, or some other Agile method).** Agile methods are actually a collection of software development practices that share a common set of defining principles but are otherwise unrelated and at times contradictory. You can learn more about what Agile really is from the Agile Alliance.
- **Myth #2: Agile methods can't work for large groups.** This statement is absurd. Agile is a collection of disparate methods. Some of those methods won't work for large groups, some will, and some can if you get creative. You have to study the specific method in question before jumping to inane conclusions.
- **Myth #3: Agile methods can work for large groups.** The Agile philosophy values "customer collaboration over contract negotiation" and "responding to change over following a plan." Customer collaboration is tough with over 100 million customers. Contract negotiation is essential to manage cross-team dependencies. (See "My way or the highway—Negotiation" in Chapter 8.) Following a plan is required for business commitments because partners get touchy when millions of dollars are involved. Applying Agile methods to large-scale projects requires you to be flexible and creative to deal with these issues.
- **Myth #4: Agile means no documentation.** The Agile philosophy values "working software over comprehensive documentation." Many Agile zealots read this and say, "Yay, no documentation!" If you think the world ends where your hallway ends, you don't deserve a cut of revenue generated beyond your walls. The Agile philosophy states, "While there is value in the items on the right, we value the items on the left more." In other words, working software is valued more than documentation, but essential documentation is still valuable for customers, partners, and cross-group dependencies.
- **Myth #5: Agile means no up-front design.** The Agile philosophy values "responding to change over following a plan." Many Agile zealots misinterpret this to mean, "No need to think or plan; the design will just emerge!" Emerge from what—a radioactive sewage dump? The point is to value responding to change over taking your original plans too seriously—it's not to jump off a cliff and see what happens next.

- **Myth #6: Agile means no individual accountability.** The Agile philosophy values "individuals and interactions over processes and tools" and "responding to change over following a plan." Many terrified managers think this means zero accountability. In fact, Agile has an interesting twist in this area. Agile makes the individual accountable to the team and the team accountable to management. Accountability is strongly emphasized, but the extra level of indirection allows Agile teams to be more efficient, resilient, and…well,…agile.
- **Myth #7: Scrum is an acronym.** This is a silly myth, but it drives me crazy. Scrum is one of the best-known and most widely practiced Agile methods, but it is not an acronym. Scrum is named after the rugby term that describes when the teams get together, arms latched in a circle, trying to obtain possession of the ball. It also is the name of the daily standup meeting used by Scrum teams. At Microsoft, we've been using a form of Scrum for decades—well before the term existed. It is one of the simplest Agile methods and the closest to what many Microsoft teams already practice. More on Scrum later.

Get the rules straightened out

Talking about Agile in the abstract makes for entertaining debate, but applying it is where the action is. Because we've established that Agile is actually a collection of software development practices, the question remains, "Which ones work well in large-scale projects?" Many people have thought and written about this question—but many people don't write this column. Before I give my opinion, some ground rules…

- **No change for change's sake.** If a team is already working well by all the measures the business cares about, there's no need to change. Change is costly no matter how nice the result might be. You should change only to eventually improve. So if no improvement is needed, no change is needed.
- **Don't get carried away.** If change is needed, don't change everything at once. Have feature teams pick one or two improvements each and see how that goes. Not every team needs to change simultaneously, and not every team needs to change identically. Of course, if you are changing a central service, like the build system, then all teams will eventually need to adopt it. But even those kinds of changes can be either spread out or made transparent to individual teams. The idea is this: try a little, learn a little, and then try a little more.
- **Differentiate between the project level and the feature level.** The biggest area where people get confused—particularly with Agile methods—is differentiating between the project level and the feature level. At the project level, you need firm dates and firm agreements between teams. At the feature level, you…well actually,… whatever. That's the bizarre idea many managers fail to understand—your team can hit whatever date you care to set; the question is only what features you end up including.

As long as the project-level plan can be tracked and followed, your feature teams should choose whatever method allows them to be the most effective.

Eric Aside This is a power-packed paragraph. One caveat: groups generally work better when the small teams within them are using similar methods. The methods needn't be identical, but teams will work best together if they have the same pacing. Otherwise, coordination and communication get muddled between teams because they have different expectations around timing.

Ready for something different?

So you're thinking about trying Agile—or perhaps you just want to placate the Agile maniacs in your group with Scrum snacks to go with the hypnotic Kool-Aid the maniacs are drinking. What should you try, and how can you best integrate it into common practices? There are a large number of Agile methods, so I'll address only the most popular ones: Scrum, eXtreme Programming, Test-Driven Development, pair programming, user stories, refactoring, and continuous integration.

First, there are two methods that we've been using for more than a decade at Microsoft: refactoring and continuous integration. Refactoring is simply reorganizing your code without changing what it does. Refactoring is used to break up complex functions (spaghetti code) or to add new functionality to existing code—like changing a class that reads CSV files into an abstract class that could read CSV or XML files. Continuous integration is the philosophy of always integrating new code into regular, ideally daily, full builds so that everyone can test it.

Let the man speak

Next are user stories, which are like a combination of scenarios and one-page specs. The idea of user stories is to provide just enough information to be able to estimate what it would take to implement and test the functionality specified.

The difficulty with user stories is that they are supposed to be written by the user. Many Agile methods assume that the user can regularly hang around with the feature team. Unfortunately, that presents a problem when you've got 100 million users.

Like it or not, we need proxies for users. Groups like marketing, product planning, user experience, sales, and support can play that role. Their findings can be codified in value propositions and vision documents that draw from a broad collection of user research. However, as those broad visions and end-to-end scenarios are broken down, we can still use the concept of user stories at the feature level to provide just enough documentation to estimate the implementation and verification of a feature set.

You complete me

Pair programming involves two people sharing a desk and a keyboard and coding together. The idea is that as one person is typing, the other is seeing the bigger picture and catching suboptimal design or implementation. The pair switches off from time to time. While two heads are better than one, they also cost twice as much. I'd rather see the two heads be put to better use in design and code inspections. However, pair programming is great for getting people up to speed in new code bases by pairing developers familiar and unfamiliar with the code.

Eric Aside My team has since adopted pairing for creating new content in unfamiliar areas. It works extremely well.

Aside from refactoring and continuous integration, Test-Driven Development (TDD) and Scrum have proved to be the easiest and most effective Agile methods applied at Microsoft. As I described in my column on Lean engineering (earlier in this chapter), in TDD you start with a class definition of functions or methods and then write unit tests for the public functions and methods before you write the code. It is an iterative procedure, in which you are writing only a few unit tests and a little code at a time. The technique is popular because it gives developers the unit test code coverage they need while producing a minimal, yet high-quality, implementation design.

It's also more fun to write the tests first. When you write the code first, the unit tests are a pain to retrofit and will only give you bad news—not exactly reinforcing. When you write unit tests first, it's easy to fit the code to the tests and you feel vindicated when tests pass.

Eric Aside While I agree with many practitioners that TDD's true purpose is exceptional implementation design, the benefits of positive reinforcement for unit testing cannot be overstated.

TDD can be used in conjunction with pair programming by having one developer write a few tests and the other implement enough code to make the tests pass, switching off from time to time. Finally, TDD gives developers a clear sense of when they are done with the implementation: when all requirements have tests and those tests pass.

A bit extreme

eXtreme Programming is a whole development methodology. It combines user stories, pair programming, TDD, refactoring, continuous integration, and a bunch of other practices into a coherent set. It is ideally applied by small teams working closely with their customers.

eXtreme Programming relies a great deal on team knowledge and direct customer interaction, using almost no documentation. This is great if your team is isolated and your customers are down the hall, but that's not exactly common at Microsoft. Our situation would be tragic if not for the billions of dollars we earn every year. However, as I've already mentioned, many of the individual methods used within eXtreme Programming apply nicely to our product development.

Are you ready for some rugby!

The last and perhaps most misunderstood Agile method I'll cover is Scrum. Aside from people confusing Scrum with eXtreme Programming (which doesn't really use Scrum) or thinking that Agile equals Scrum (huh?), the most bewildering part of Scrum is all the strange terms associated with it: Scrum Masters, backlogs, burn-downs, sprints, and even pigs and chickens. It's enough to scare any manager away. Big mistake.

For better or worse, Scrum was invented by a person who enjoys funny names and stories. The practice itself is neither complicated nor contentious. So, aside from refactoring and continuous integration, Scrum is the closest Agile method to what we've been doing internally for years, with a few significant improvements.

Let's start by mapping some of the confusing terms. Scrums are daily stand-up meetings, Scrum Masters are feature team organizers, backlogs are feature or work-item lists, burn-downs are graphs of remaining work, sprints are mini milestones, and pigs and chickens are entrepreneurial farm animals (long story, cute joke).

None of these concepts are new, but Scrum does introduce some big improvements:

- The daily stand-up meetings in Scrum are highly organized and collect useful data. The team organizer (Scrum Master) simply asks all the team members what they accomplished since yesterday (and how long it took), what they are working on until tomorrow (and how much is left to do), and what's impeding progress.

  Eric Aside Tracking how long it took is my team's small contribution to Scrum at Microsoft. By adding this information to the burn-down data (how much is left to do), you can produce fantastic cumulative flow diagrams, measure time on task and work in progress, and better estimate team capacity. Typical time on task is around 42% for production teams; 30% for teams focused on communication—like mine—and as much as 60% for co-located feature teams.

- The data collected at scrums is entered into a spreadsheet or database. From the spreadsheet, you can analyze time on task, completion dates, work in progress, plan changes, and a whole host of project issues. One of the most popular graphs is a burn-down chart that plots time vs. total work remaining.

Online Materials Sprint Backlog (SprintBacklogExample.xls; SprintBacklogTemplate.xlt)

- The Scrum Master is an independent force on the team. It's best if he or she isn't even part of the group, but often that's not realistic. The Scrum Master has permission to keep meetings short and cut through the crud.
- The feature list or schedule is called the *Product Backlog*, and the work-item list or schedule is called the *Sprint Backlog*. By keeping these two lists separate, management can focus on the work they want done (the Product Backlog) while the team focuses on the work at hand (the Sprint Backlog). Typically once a week, the Scrum Master meets with management (for example, at the weekly lead's meeting) and updates status, ensuring everything stays on track.

Online Materials Product Backlog (ProductBacklogExample.xls; ProductBacklogTemplate.xlt) and Sprint Backlog (SprintBacklogExample.xls; SprintBacklogTemplate.xlt)

- Sprints, the mini milestones, are fixed in length. They are over when the specified number of days is over—typically around 30 days.

Eric Aside In the six years since writing this column, I've been part of teams that used 1-week sprints, 2-week sprints, and 30-day sprints. Right now, my team uses 2-week sprints, which are my favorite. Two weeks is short enough not to require much overhead—everything can be tracked on a whiteboard. However, two weeks is long enough to get serious work completed. I'm now looking forward to trying Kanban and being on almost a continuous sprint, but that's another story.

- After every sprint, the feature team reviews its work with management (nice change, huh?); discusses what went well and improvements for the next sprint (a wee bit better than waiting a year or decade until the post-ship postmortem); and plans and re-estimates the work items for its next sprint (the plan and estimates changed? No way!).

By using daily, weekly, and monthly feedback mechanisms, Scrum allows teams to work efficiently and resiliently in a changing environment. By collecting a little key data, Scrum allows teams and management to know how teams are operating and to spot issues before they become problems. By separating the feature list owned by management from the work-item list owned by the feature team, Scrum allows teams to direct themselves and stay focused. It drives accountability to each member within the team and to management outside the team.

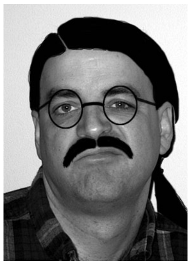
The more you know

Not all Agile methods are for everyone, and many won't work on big Microsoft projects. But Scrum, Test-Driven Development, refactoring, and continuous integration are being used by many Microsoft teams with great effects. Pair programming and user stories are being applied to a lesser degree, but they can be effective in the right situations. As long as you don't get carried away and start forcing Agile down your team's throat, there's a great deal to be gained by applying these methods.

Eric Aside Managers have forced methodology changes down engineers' throats almost everywhere I've been employed. It never works, even for something popular like Scrum. Managers can suggest, support, and subsidize behavioral change, but they should never coerce it.

To learn more, search for Agile methods on our internal network or on the web. Also watch for new courses on Agile methods coming in spring 2006. If everything is going great on your team, then don't change a thing. But if you'd like to see a little higher quality or better feature-team project management, you owe it to yourself to take some antacid and give Agile a try.

October 1, 2007: "How do you measure yourself?"



At Microsoft, we can execute, but can we think? When billions of dollars are on the line, you better not be guessing about decisions. A decade ago, our products weren't guesses; they were enhanced impersonations of our competitors' successful products. We won by outdoing those ahead of us.

Now we lead in many areas, and without competitive targets, brain-challenged teams rely on guesswork. Their mantra: code cool stuff and hope that customers find something they like. Their results: a disorganized mess with little value or uptake.

Thankfully, enlightened teams don't guess. They rely on data Microsoft gathers from research and directly from customers to determine real points of customer pain and delight and then enhance our products accordingly. Without that data and feedback, we'd be hopeless. Yet just mention using data to drive *how* we build our products and you'll be lucky to leave the room with your body intact.

There is no try

If great teams use data to remove guesswork from what we build, why does guesswork dominate how we build it? Today's software development process is all about guts and glory. "Best practices" are conventional wisdom, processes are tribal knowledge, and many self-righteous Agile methods are loaded with dogma instead of data. Why?

Eric Aside My favorite Agile methods, like Scrum and Test-Driven Development (TDD), are loaded with data. TDD data is more subtle—the percentage of requirements with passing tests.

Don't tell me there is data out there that proves certain methods are effective. I'm not talking about someone else's data, I'm talking about yours. How do you know your team is using the right methods with the right results before all the bugs arrive? How do you know if you're any better than yesterday? Why aren't you using data throughout to find out?

Maybe it's because software development is a creative process or a craft that can't be measured. Maybe measurements are faulty or easily gamed. Maybe there's so much data that you could use it to justify anything. Or maybe frightened fanatics are foolishly focused on regressive rationalizations of their suspect superstitions. They are too scared to measure and too ignorant to know how.

Well giving it your best shot on tried and true methods isn't good enough. Not with this much money and this many people's lives and livelihoods on the line. Being clueless about using the right metrics the right way is no way to go through life, buddy. Fortunately, you don't have to be in premed to understand it.

Is there a problem here?

I hear you saying, "You sick man, don't you know metrics are evil? Don't you know hollow-headed managers will use them to pit you against your peers, and your team against other teams doing different work? Don't you know they just get gamed, while real progress and real customers suffer?" Yeah, I know. We've already established you don't know anything about using measures properly. But since you brought it up, let's break down your objections:

- **Software is a creative craft that can't be measured.** As I talked about in "A software odyssey—From craft to engineering" (in Chapter 5), craft is fine for tables and chairs, but it's not good enough for bridges, pacemakers, and software people depend upon. Regardless, you've missed the point. **Lesson 1: Don't measure how, measure what.**

- **Measurements are faulty and easily gamed.** Others put this as, "You get what you measure." If you measure lines of code, people write lots of bad code. If you measure bugs fixed, they create more bugs to fix. **Lesson 2: Don't measure intermediate outcomes, measure desired end results.**
- **There's enough data to justify anything.** Computers produce lots of data, and software development happens on computers. However, all that data is useless if it presents more questions than it answers, regardless of how pretty the graphs might look. **Lesson 3: Don't just collect data, use measures to answer key questions.**
- **Managers will use data against you.** Managers are notoriously lazy. Why apply thought if numbers tell you what to do? Good measures don't tell you what to do, because good measures don't measure how (remember lesson 1?). **Lesson 4: Don't use measures that make your decisions, use measures that tell you a decision is needed.**
- **Managers will make unfair comparisons.** Managers are notoriously clueless. From their perch at 10,000 feet, software is software and bugs are bugs; all subtly is lost. Focusing on desired end results helps, but it's not enough to avoid improper comparison. **Lesson 5: Don't compare raw measures, use baselines and exemplars that provide needed context.**

Eric Aside My friends at Google and startups might claim, "This is easy. Get rid of managers." Nice try. Replace "managers" with "executives" or "product owners," and you've got the same issues.

Now let's follow the lesson plan.

What's going on?

Lesson 1: Don't measure how, measure what.

People hate being forced to work a particular way. Sure, they like pointers and suggestions. They can live with constraints and requirements, but no one wants to be an automaton.

Once anyone starts doing a task, they are sure to find ways to do it better. Forcing people to work your way, instead of their own, is guaranteed to hit a point where your way is worse than theirs. This leads to feelings of frustration on their part, and feelings of resentment, stupidity, and disrespect toward you.

Measuring how you want something done is equivalent to telling people how to do it. That sets you up as an idiot people resent and disrespect. I don't recommend it.

Instead, measure what you want accomplished and leave the how to the intelligent human beings doing the work. Say you want a scenario to work. Instead of measuring spec completion, function points, or bugs remaining (all hows), break down the scenario into segments and measure how many segments and segment transitions work as desired. Ideally, you'd have a customer be the judge, but an independent tester would suffice.

In the end you'll thank me

Lesson 2: Don't measure intermediate outcomes, measure desired end results.

Metrics get gamed. We all know it; most of us have done it. Why? Because managers manage metrics. If you don't hit your metrics your manager is going to emerge from his or her cave and annoy you. That makes the goal "hitting your metrics" not "hitting your goals."

How do you avoid the trap of hitting metrics instead of goals? Two ways:

- Don't use metrics; be dumb and happy.
- Make hitting your goals and your metrics equivalent.

Think about your team's goals. What are they really? What outcomes do you want as a team? What are you trying to accomplish? Measure that desired end result. Then it won't matter how the team hits those metrics (within reason), because hitting them is just what you wanted.

Eric Aside Read that paragraph again. It's amazing how many people don't get this.

I want to know right now

Hold on a second, a panicked manager has a question: "If I only measure end results, how will we ever get there?!?" That's actually a good question. No one successfully develops software without iteration—take a small step forward, check if you're on the right track, then take another step. How can you check if you're on track if you're only measuring the end result?

There are two approaches to iterative feedback that still focus on desired end results:

- **Make every iteration produce end results.** This is the best approach and a fundamental concept of Agile methods. By producing customer value each iteration, you can regularly check with customers to see if you're on track. Your metrics should enhance this effort by measuring end results the customers want (such as usability, completeness, and robustness).

- **Apply predictive measures that tightly correlate to desired end results.** This approach isn’t quite as good because correlation is never perfect. However, some end results can’t be measured accurately till the end, so using predictive measures is necessary (for examples, read “Bold predictions of quality” in Chapter 5). If you must use predictive measures, always back them up with measures of the real results to be sure you are getting what you want.

Then make your choice

Lesson 3: Don’t just collect data, use measures to answer key questions.

Software development, by its very nature, produces tons of data—build messages, test results, bug data, usability research, complier warnings, run-time errors and asserts, scheduling information (including burn-down charts), source control statistics, and on and on. Being software engineers, you’ve probably piped this data into various reporting packages and produced endless charts and graphs. Well, good for you.

Actually, is it good for you? No, no it isn’t. Too much information is just as bad as not enough. In a crowded mall, you don’t hear more conversations, you hear nothing at all. Your brain treats everything as noise and blocks it out. The same thing happens with too much data.

Collect all the data you want, but don’t throw it in people’s faces. Instead, think carefully about the desired end results you want. What key attributes do you care about? Some people call these Critical to Quality (CTQ) metrics or Key Performance Indicators (KPIs). You want a small, focused mixture of specific and generic CTQs.

Some CTQs will be specific to your product. Say you’re working on wireless networking. A desired end result is a quick and lasting connection, so your CTQs would be time to connect and average time till connection failure.

Some CTQs will be generic to software development. Say you’re lean minded (and I hope you are). You’d care about minimizing cycle time. Your CTQ would be time to complete a scenario as desired from start to finish. Say you’re engineering-quality minded (and I hope we all are). You’d care about solid, stable code. Your CTQs would be a predictive measure, like code churn or complexity, and an actual measure, like Watson hits.

Eric Aside *Watson* is the internal name for the functionality behind the Send Error Report dialog box you see when an application running on Windows crashes. (Always send it; we truly pay attention.)

We are in charge

Lesson 4: Don't use measures that make your decisions, use measures that tell you a decision is needed.

I think the biggest fear any employee has when it comes to metrics is being treated like a number. I wrote about the pitfalls of this in my column "More than a number—Productivity" (see Chapter 9). If your review and rewards come down to a formula, something is seriously wrong.

The same goes for all decisions. If the decision comes down to a formula, then all thinking and consideration are absent and we become servants to our processes and tools. That is backward. Processes and tools work for us, we don't work for them.

Luckily, if you follow the previous lessons, measuring only your desired end results, your management can't use the measures to make decisions. Yes, if you consistently missed your team's desired results, management could make you suffer, and you'd deserve it. However, because all management has are the end results, they wouldn't know why you missed the results or who or what was responsible. They'd have to investigate, understand, and analyze. They'd have to think before coming to a conclusion.

Great metrics tell you you've got a problem. They can't and shouldn't tell you why. Root cause analysis requires careful study. If people say the easy answer is in a metric, both they and the metric are lying.

A girl's gotta have her standards

Lesson 5: Don't compare raw measures, use baselines and exemplars that provide needed context.

Wait, a panicked engineer has a question: "Okay, so we measure a great end result, like completed scenarios, and our feature team has half the number of scenarios completed as another team. Now our manager is demanding we work harder and longer, even though our scenarios were broader and far more complex. Using the 'right' metric is only causing us grief!" That's a fair point. I've got good news and bad news.

The good news is that the manager is actually trying to fix a real problem (the right metric helped). The bad news is that the manager didn't consider what the problem was. Instead of analyzing the root cause of the problem (complex and broad scenarios), the manager is assuming the problem is lazy engineers. You need to help your manger by providing context.

The easiest and best forms of context are baselines and exemplars.

- **Baselines tell you what to expect from a metric.** The first time you get an end result, its measure is the baseline. From that point forward, you know if you are getting better or worse by comparison with the baseline. Your manager can't be surprised your scenarios are broad and complex if your baseline already established that fact. Baselines are extraordinarily handy for tracking improvement.

- **Exemplars tell you how good your results could be.** The best result achieved for a measure is the exemplar. It doesn't matter how it was achieved or which team achieved it. The difference between your results and the exemplar is your opportunity to improve. "But what if they cheated to get scenarios done faster?" If done meets the quality and compliance bar, then they didn't cheat. They just found a better way. "But what if our scenarios are broader and more complex?" Well, you should break them down and simplify. Remember, you are measuring desired end results. If you really care about delivering value to customers in fast, small chunks, you need to keep your chunks fast and small. Exemplars are priceless for spotting your biggest improvement opportunities.

A unique perspective on the world

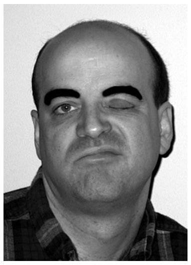
So, now you know what and how to measure and the differences between good and bad metrics. You also know ignoring metrics leaves you happy but dumb. Ignoring metrics turns software development into guesswork and leaves your success to chance. I believe the polite word for such irresponsible behavior is "foolishness."

However, you've probably also noticed that good metrics that measure only desired end results are not generic. You can't simply use the same ones for every project. Sure, there are some engineering quality and efficiency results you always want (like being productive, secure, robust, and responsible), but other results around performance, usability, and overall customer value depend on your desired scenarios and the customer's needs.

This means that putting the right measures in place isn't trivial. It requires some thinking as a team to decide what you really care about for this release and how you'll know you've reached your goals. Then you'll need to make those measurements part of your iteration and feedback process from the beginning to always know you're moving in the right direction. Heresy, right? Actually working toward known goals? Maybe I'm the fool to think we'd be so sensible.

Eric Aside Personally, I believe the test discipline plays a huge role in defining and tracking metrics. They tend to love data and look at using software from the customer's perspective. The key is for test to focus on measuring desired end results, not create more charts and noise.

October 1, 2010: "You can depend on me"



We're getting into the end game before a big release, and I'm already tired of people whining about unstable and overdue dependencies. Of course they are unstable and overdue, what planet are you from?

Yeah, yeah, a package should only depend upon packages that are more stable than it is (the Stable Dependencies Principle). I've pushed this principle countless times. Yet when you work for a big ambitious technology company like Microsoft, no one wants to wait for cool technology to stabilize before coding against it—at least no executive I've ever met.

That means your dependencies are unstable and likely running late. It's not the fault of the teams you depend upon, and it's not going to be much better next time. Tough luck—quit whining and deal with it. Don't know how? I figured.

Eric Aside This is the first of a series of four process improvement columns I wrote in five months, after not writing any for three years. Why the hiatus and then sudden flood of thoughts? Because seven months prior to this column I switched back into a product group as the development manager for the Xbox.com websites. I had been training development experts, leads, and managers for a while, but now I was back doing it myself.

Upon returning to being a development manager, my first columns were about either coming up to speed in a new role or topics that were on my backlog and still fresh in my mind. Once I was on the job for six months, all the annoying inefficiencies started grating on me, and I returned to focusing on engineering improvements.

Amongst our weaponry are

There are five methods of dealing with unstable dependencies.

1. Convert them from hard dependencies to soft dependencies or knowledge dependencies.
2. Over communicate and project manage the heck out of them.
3. Get as close as possible to them personally, physically, logistically, and logically.
4. Automate ingestion of their work for you and testing of their work for them.
5. Create multirelease plans, stable interfaces, realistic schedules, and a vision that leads instead of chases.

Hold on, that last method is a pipe dream—there are four plausible methods of dealing with unstable dependencies. Let's break them down.

Eric Aside Unstable and overdue dependencies are avoidable by creating multirelease plans, stable interfaces, realistic schedules, and a vision that leads instead of chases. The teams at Microsoft (and elsewhere) that have figured this out live better lives and deliver great, dependable experiences on a predictable schedule.

These thoughtful teams sacrifice a little bit on timely innovation. However, keep in mind that Apple is widely considered a highly innovative company, yet Apple's innovations don't utilize bleeding-edge technology. Instead, they craft innovative new experiences from proven technology.

I think your brain is going soft

A hard dependency is one that you literally can't ship without. If it fails, you fail. A soft dependency is a dependency with a fallback position. If it fails, you can still ship with reduced functionality.

Unstable hard dependencies are a recipe for panic followed by disaster. You want to convert them to soft dependencies by agreeing to a fallback plan. Typically, fallback plans involve shipping with a previous version of the dependency, reducing functionality, taking ownership of the dependency's module, or some combination.

Fallback plans are wonderful psychologically. They remove the fear and uncertainty around failure. Everyone knows what will happen, and it doesn't involve bloodshed—only lackluster reviews and a less compelling release. Everyone is still motivated to deliver something great. With raw fear off the table, people collaborate and problem-solve far better.

Taking a snapshot of your partner's code converts a hard or soft dependency to a knowledge dependency. You aren't actually dependent on the other team for anything but its knowledge and past experience.

Knowledge dependencies are underutilized—they don't get the respect they deserve. Just because your team may not want to take on any hard or soft dependencies doesn't mean you can't take advantage of the knowledge and experience of people who've done something similar before. I talked about this in "NIHilism and other innovation poison" in Chapter 10.

Failure to communicate

When you are dealing with overlapping and overcommitted schedules, like when you're working on almost any project ever, you need to over communicate to your partners and project manage them. It doesn't matter how reliable they are or how well coordinated you appear to be. Assumptions will be made, and important details will get missed. You need to say everything to everyone regularly and repeatedly, and track every deliverable.

You'd think all this extra communication would become noise, but it doesn't when handled properly—with regular face-to-face meetings, item tracking (think Product Studio or TFS), and formal e-mail for plan changes.

- Regular face-to-face meetings (once a week or so) are great for coordinating small changes, fixing issues that arise, and doing all-important sanity checks. A sanity check is five minutes spent validating high-level assumptions. ("We're still getting these key deliverables in two weeks, right? You're still gainfully employed, right?")
- Item tracking in a work-item database, like Product Studio, TFS, or any number of other commercial packages, is perfect for tracking resolution of bugs and work items across teams. Share the database queries you use with your partners so that everyone sees the same status.
- When your or your partners' plans change, everyone needs to know. Start with a formal e-mail to everyone involved (Scrum Masters, leads, managers, and directly impacted team members). If any work items have been dropped, changed, or added, update the work-item database accordingly. Follow up at the next face-to-face meeting with a full description of what changed and why it changed. This would appear to be obvious, but one person's big change is another person's minor detail. That's why you also do sanity checks.

Eric Aside Clearly this extra communication and project management is extra work. So is everything else in this column. The extra work typically hits program managers and testers the hardest, but developers are also impacted. The amount of extra work is proportional to the type of dependency (hard, soft, or knowledge) and the level of associated chaos. Plan accordingly.

We two are one

The easiest way to stay in close contact and resolve issues quickly is to practically join teams.

- **Personally** Get to know your partners personally. Meet together, socialize together, and truly understand each other. A good working relationship helps in all sorts of ways. You become committed to each other's success.
- **Physically** Sit with your partners physically. The whole team probably won't fit, but having one or two individuals spending significant time in your partners' space will do wonders for catching issues early on both sides.
- **Logistically** Tie yourself to your partners logistically. When they deploy, you deploy. When they beta, you beta. When they ship, you ship. Staying in sync will save you oodles of trouble—trust me on this.

Eric Aside Being flexible and agile really pays off in this case. Using short iterations and always being ready to ship not only helps you minimize work in progress and reduce technical debt, it also helps you stay synchronized with your partners' releases.

- **Logically** Engage with your partners' tools, work-item databases, and source code. The deeper you know what's really going on in their work, the better you'll foresee, understand, and resolve issues.

Eric Aside Even if your partners haven't finalized their interfaces, a starter interface can often help you get an early jump on development and testing. You can write your own emulator, use an early drop from your partners, and otherwise code and test against the upcoming interface in advance of receiving the final version.

It's totally automatic

One critical engagement with your partners' tools is around handoffs. Before they deploy a new version, they should run a set of build verification tests YOU WROTE—only you know what you are expecting from your partners. After they deploy a new version, you should run a set of ingestion tools THEY WROTE—only they know all the moving parts, tricky ordering, and special steps necessary.

The build verification tests you wrote should quickly check that the new version works the way you intend to use it. Writing these tests can be a bit tricky because you have to understand your usage patterns, and you have to author the tests in their test system. Of course, all that effort is well worth it when every handoff works as expected.

The ingestion tools they wrote should give you all you need to use the new version. This should include setup, libraries, content, configuration, and validation. Writing these ingestion tools shouldn't be wasted effort since they help your partners as much as they help you. That said, all the effort is well worth it when they don't spend two days after every handoff getting your systems functioning again.

Eric Aside We are finally putting some of these tools in place on my team. It's so much better.

No whining!

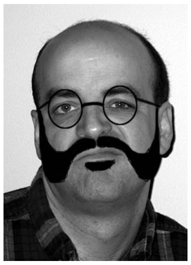
Even in the best of circumstances there are always surprises in any development cycle. Being flexible, using short cycles to react faster, and communicating well with partners, customers, and within your own team does wonders for dealing with the unexpected.

What isn't constructive is blaming your partners for costly mistakes, even if they were at fault. We're all human and mess up occasionally. We win and lose together. If you can't handle the problem or didn't know about it in time, then you're at fault too. You all can improve your communication and issue management next time—the issue will recur.

For all the headaches and heartaches that unstable dependencies can cause, they also can be exciting, build a larger sense of team, and bring faster, broader, and bolder innovation to customers.

Don't play the victim. Create fallback plans, over communicate, integrate your teams, and automate quality handoffs. You can be part of something big if you embrace the challenge.

November 1, 2010: "Am I bugging you? Bug Reports"



Some developers hate seeing bugs. They think bugs indicate a failure on their part—that their code seemed perfect until bugs were found. These developers are called "amateurs." Real developers know the only reason you haven't found bugs is that you haven't looked.

I love seeing bugs. It's better for me to see them than for my customers to see them. What I hate seeing are poorly written bug reports—misleading or generic titles, unclear or missing reproduction steps, exaggerated priority, overstated severity, and inappropriate, cowardly, and poorly documented resolutions.

Why can't people write decent bug reports? It's not like a decent report is longer or much harder to write than a lame report. It's not like clear definitions for everything in a bug report don't exist. Ah, but those definitions do vary and sometimes conflict from team to team. What are the right definitions to use for everything in a bug report? I'm glad you asked.

Eric Aside Every piece of software ever written has hundreds or thousands of bugs, depending on its size and complexity. Some bugs are innocuous, like "I'd prefer the close button to be wider." Some bugs are misunderstandings, like "It wouldn't let me use an obscene name for my gamer tag." And some bugs are nasty issues that must be fixed no matter what, like exposing personal information. Since bugs are often found by people outside the development team, bug reports must be written and tracked to closure—typically using a work-item tracking database, like Product Studio (a legacy MS tool) or Team Foundation Server.

Bug dissection

All bug reports have the same basic set of information.

- **Title** A short description of the issue
- **Assignment** Who’s taking care of the issue at the moment
- **Repro steps** The steps necessary to reproduce the problem
- **Priority** The urgency and importance of the issue
- **Severity** The fallout from the issue
- **Resolution** How the issue was resolved

There are a bunch of other fields that are helpful in reproducing the issue and understanding the root cause, but the basic set is short and simple. Let’s cut through the controversy and lay out the rules for each field.

Title and assignment

The title of a bug should be a terse, one-line description of the issue that is specific enough to identify that issue uniquely, making bug report searches and identification easy. “The screen blanks when you hit the Cancel button” is a poor title. “Blank screen after canceling avatar editor” is a great title. The second version is shorter, yet provides more specific context around where and when the issue occurs.

When you create a new bug report, you must assign it to someone to resolve. However, unless you are part of the development team, you shouldn’t assign the bug to an individual, even if you know the whole team personally. Instead, assign the bug to the team. This is typically done by specifying the area or team in the bug report and accepting the default assignment. The default assignment is typically “Active” or “Triage.” You don’t know better. Trust the team to know who should work on the issue.

Eric Aside There are some groups that want all bugs assigned to individuals. This ensures no bugs are ignored. However, even those teams must check for bugs assigned to Active or Triage to ensure they aren’t missed. After all, people outside the team don’t know what other values to use.

As a general rule, all bugs should be assigned to individuals or groups that will check them regularly. Since most triage teams meet daily, I’ve always liked the idea of assigning bugs to Active or Triage as a default.

Repro steps

There is nothing more frustrating than a bug report without a decent repro (reproduction steps). It's like your significant other telling you, "You know what you did!" with no further explanation. Now I know I messed up and have no way to correct it. Terrific.

Repro steps should be short and sweet—the minimal set that triggers the issue. You should also include the build number (typically a separate field), the environment you used (the version of the operating system, browser, and any other relevant details), and any preparation necessary (like signing into Xbox.com with a gold account).

Sometimes you aren't sure how you triggered the bug because it's intermittent or associated with a weird state. In this case, provide the build number, environment, and setup, then describe the circumstances, acknowledging that precise repro steps aren't clear.

Eric Aside We have many internal tools, like Watson and Autobug, that generate bug reports automatically. Naturally, these tools have some limitations in producing repro steps, but they can often still supply stack traces, build numbers, environment information, and other details that help isolate issues.

After describing the minimal repro, you must indicate what you expected to happen ("Expected"), followed by what actually happened ("Actual"). All repro steps should have these three sections—the setup, the expected results, and the actual results. That way someone reading the bug report knows exactly what went wrong and how to reproduce it.

Often a picture or video tells a thousand words. There are many tools to create screen captures of both stills and compressed videos. Attaching these files to a bug report can be the difference between a properly fixed issue and an elusive one.

Eric Aside It's annoying to see a bug report with 15 repro steps when the issue can be reproduced in 4 steps. Not only are 4 steps shorter and easier to understand, but they also allow the developer and the tester to close the bug far faster. It takes less time to reproduce the bug, less time to determine the cause (fewer possibilities), and less time to verify the issue has been fixed.

Priority

There are endless arguments over the meaning of the Priority field, a value that typically ranges from 0 to 3. Surely you have better ways to spend your time. Instead, let's lay out a few simple rules, and then define priority based on those rules.

- Priority should never have to be adjusted once properly set, unless the bug itself changes character. If priority 1 means "fix this sprint or milestone" and priority 2 means "fix next sprint or milestone," then you've got to change the priority of bugs at the end of every sprint or milestone. Not only is that a waste of time, but it updates the "last changed date" on the bug, an act that causes the loss of important information.

- Priority should be easy to assign and differentiate. You don't want the team spending a bunch of time arguing over the priority of every bug. It should be obvious, both when writing the bug report and when reading it.
- Priority should be memorable and actionable. No one should have to ask, "What was pri 2 again?" No one should have to question what needs to be done for each priority level.

Based on these three rules, here are priority definitions that serve well.

Priority	Description	Timeframe
Pri 0	A CRITICAL failure that requires URGENT attention that doesn't have a known WORKAROUND. This is a blocking bug.	You can use the bathroom after you resolve the issue or find a workaround.
Pri 1	A CRITICAL failure that requires URGENT attention.	Must be resolved in the current sprint or milestone.
Pri 2	A CRITICAL failure.	Must be resolved before release.
Pri 3	A failure or suggestion.	Should be resolved before release.

Pri 0 issues typically block testing, deployment, or some other time-sensitive work. Given the seriousness of pri 0 bugs, you can't submit them and expect something to happen. You must send mail to the individual or team and then call or walk over and talk to them until someone is actively working on resolving the issue. If a viable workaround is found, pri 0 bugs should be changed to pri 1.

Eric Aside Teams do vary their definitions of priority. Some start at pri 1 instead or pri 0. Some break the rules I listed at the start of this section or have a separate field to indicate a blocking bug.

If you open a bug in a different team's work-item database, be sure to use their definitions. The definitions typically pop up in a tooltip or help screen.

Severity

Severity is even simpler than priority, yet it's also often misused. Severity refers to the fallout of the issue, NOT how important it is. The definitions are:

- **Severity 1** The issue causes a CRASH or customer data LOSS
- **Severity 2** The issue causes a MALFUNCTION that inhibits action
- **Severity 3** The issue causes an INCONVENIENCE or unfinished LOOK

Please note that severity is independent of priority—in other words, severity has nothing to do with priority. A priority 1 bug is more important than a priority 2 bug, regardless of severity. Displaying offensive content is severity 3 but priority 1. Crashing when a user does a forced reboot is severity 1 but priority 3. Nothing makes you the subject of engineer ridicule like claiming a noncrashing bug is severity 1 just because it's high priority. You sound like an idiot.

Resolution

One of the most important and most often misused fields in a bug report is Resolution—the indicator of what was done to resolve the issue. Resolving a bug means you are no longer concerned about the issue and you don't plan any further work once the bug originator verifies that the resolution closes the bug.

If the issue requires more work before you release, even if it's not the responsibility of your team, then the bug should remain active and assigned to one of your team members to track.

Here are the possible values for the Resolution field in alphabetical order:

- **By Design** The bug report describes the intended behavior. It works as designed.
- **Duplicate** The bug has the same cause and nearly the same user experience as an earlier reported bug. Never resolve an older bug as a duplicate of a newer bug—regardless of how much nicer the newer bug report is—unless you like making enemies of the originator and losing the "first-seen" date.
- **External** The bug is caused by something outside your control AND you can release without the bug being fixed. If you can't release without having someone outside your group fix the issue, then the bug should remain active and assigned to someone in your group to track, linking to the issue on the other team.
- **Fixed** The bug is fixed. My favorite resolution.
- **Not Repro** You couldn't get the bug to recur in the build and environment noted. Saying "It works on my machine" doesn't cut it—check with the originator first whenever possible.
- **Postponed** You won't fix this bug in this release. Postponed is for the same gutless slackers who say they'll start writing unit tests tomorrow. Real engineers leave the bug active and use a Fix By field in the bug report to indicate a future release when they truly intend to fix the issue.

- **Won’t Fix** You won’t fix the bug ever. My second favorite resolution—it shows you have enough experience to know when a bug simply isn’t worth fixing, usually because the fix causes more trouble than the bug itself.

When you resolve a bug, you must provide a description as well as fill in the Resolution field. That description is important. You get fewer arguments about resolutions, understand the issue better upon recurrence, and protect yourself and the company if the issue later makes headlines. This happened to an old team of mine once—we saved the company millions in penalties when our resolution description for an offensive content bug proved our lack of malice.

When a bug is resolved, it is automatically assigned to the person who opened it. If that person isn’t on the team, the bug should be assigned to another team member who can verify the resolution with the originator of the bug. You can’t always count on people outside the team to validate a resolution in a thorough and timely manner. Of course, if the resolution isn’t satisfactory, the bug should be reactivated.

Eric Aside I first defined resolutions for my development team 10 years ago. Looking back at that mail, the definitions here still stand.

Keep it simple

There are many other fields in a bug report. I mentioned using the Build and Environment fields to capture reproduction information and the Fix By field to indicate when a bug will be addressed. There are also fields to track root cause, how the bug was discovered, area of the product or service where the bug occurred, potential security impact, and countless other variations of information.

When setting bug report requirements, demand no less than what you need and no more than what you’ll use. Requiring more than necessary will cause people to complain and stop submitting bug reports—neither of which serves you or your customers well.

By keeping bug reports easy to write and easy to read, you encourage people to submit clear bug reports for the issues they find. Using bug templates that prefill some fields also helps. There’s no better gift to our engineers and the customers we care about than a well-written bug report that averts an issue before it ever reaches our users.

December 1, 2010: “There’s no place like production”

As much as I love Microsoft, and as many advantages as we have as a company in the intelligence of our people, the breadth of our products, and the boldness of our vision, there are times when people here are frigging clueless. It’s not everyone—Microsoft is a wildly diverse company. But there’s just enough ignorance to drive you insane.

A great example of nerve-racking naïveté surrounds our service environments. My current team has separate environments for development, check-in testing, scenario testing, stress testing, cross-division integration, partner integration, certification, and production. That’s eight different environments—and we’re planning to build out a preproduction environment next year. Here’s the punch line of this pathetic joke: even with all these environments, there are a ton of issues and scenarios that can only be exposed in production.

Why are we being so witlessly wasteful? Because we can afford it (good situation but bad reason), and because there are so many old-school enterprise engineers who don’t understand the most basic truth about services: there’s no place like production. These engineers conjure requirements for testing and integration environments based on hard-won lessons from business software, yet they fail to fathom their folly. Close your eyes, tap Ctrl-Alt-Delete together three times, and think to yourself, “There’s no place like production. There’s no place like production. There’s no place like production.”

Eric Aside Even though I wrote this column recently, it has become among my most influential and commonly quoted columns at Microsoft. I’ve seen e-mail threads from multiple divisions at director levels discussing the notions at length or simply stating “There’s no place like production.” It warms my heart.

How did I get here?

What kind of fools build out and maintain useless environments? The kind who got burned building enterprise software.

Large businesses rely on enterprise software—it’s got to work or they won’t buy it. Once they buy it, they own it. You don’t get to fix enterprise software anytime you want. That’s right, not even with security patches.

Remember, enterprise paychecks depend on having the software run smoothly. Software changes represent risk to an enterprise business. If the software doesn’t work, work well, and continue working well, enterprise businesses aren't buying it. And they'll tell you when they are darn well ready to accept a patch.

An entire generation of Microsoft engineers learned the hard way that you can't release software until the code is fully tested. There are no "retries" in enterprise software.

Enterprise engineers heave at the thought of releasing code that hasn't been fully vetted into production environments. They'd burst into convulsions if they understood the real truth about services.

Surely, you can't be serious?

What is the real truth about software services? There's no place like production.

Let's break down these myths about testing and integration environments one at a time.

- **If your check-in tests pass in one environment, they'll pass in all environments.** Okay, that one obviously is wrong, but here's what's worse. It's not difficult to write critical check-in tests that pass in production but fail everywhere else (like tests of broad fan-out or database mirroring). Instead of kidding yourself, write a small set of automated sanity checks that developers can run quickly in their development environment before they check in.
- **You need a separate environment to test scenarios before integrating code with partners.** There are two reasons people believe this—they don't want unstable code to break their partners' code, and they don't want their partners' unstable code to block testing. The first reason is perfectly rational—you need a test environment to do preliminary acceptance and stress testing, especially for critical components. The second reason is laughable—like your partners are actually going to maintain your test environment in some working state. They won't. They can't. (More below.)
- **You can't use production for stress testing.** Why not? Are you worried production will fall over? Wouldn't you want to know? Isn't that the whole point? Wouldn't it be great to watch that happens in a controlled way and back off as needed? Hello?
- **You need integration environments to check cross-division scenarios prior to release and provide preproduction access to external partners.** Assume cross-division scenarios worked perfectly prior to production. Assume external partners signed-off in a separate environment before release. Do you now have quality assurance? No. None. Scenarios don't work the same in production, where there are more machines, different load conditions, different routing and load balancing, different configurations, different settings, different data and certificates, different OS setups

and patches, different networking, and different hardware. You'll catch some integration issues, but not enough to make this enormous expense worthwhile.

> **Eric Aside** Does a virtual cloud environment, like Azure, take care of these issues? No, it only resolves the different OS setups and patches and different hardware. It helps a bit with the other issues, but only production is production.

- **You need a protected certification environment.** Why do you certify products in advance? Because you want to ensure they'll work in production. Oh wait.

Let's recap. There's no place like production. You need a development environment to run a small set of automated check-in tests, a test environment to run preliminary acceptance and stress testing to help avoid catastrophic failures, and production. Anything more is superfluous.

> **Eric Aside** It's nice for your partners to provide "one-boxes" for you to use with your dev and test environments. One-boxes are preconfigured virtual machines that run the services you depend on in a compressed image. Of course, one-boxes are nothing like production.

Then it's hopeless

"Wait a minute! We can't throw untested code at customers. They'll plotz! And don't get me started about exposing prerelease, uncertified, partner code. Have you lost your mind?!" Shut up and grow up. There's no place like production. The problem becomes configuring production to permit the testing and certifying of prerelease code.

The solution is called "continuous deployment." The concept is simple: deploy multiple builds to production, and use custom routing to direct traffic as desired. It's like a source control system for regulating services instead of source files. That it isn't built into Azure and other cloud systems is inconceivable.

There are a variety of different approaches to continuous deployment, which basically differ in regard to the sophistication of the deployment system and custom routing. However, continuous deployment can be quite simple to achieve.

The toughest part is dealing with data, which must function properly across multiple builds. However, if a service is designed to handle at least one rollback after a new build is deployed, even if that new build introduces new data, then that service will function well in a continuous deployment environment.

Eric Aside You also need to worry about variations of settings across builds. This is a little tricky, but not too bad. Ideally, your settings aren't changing all the time.

If new builds depend on new versions of the .NET Framework or the operating system, those have to be hosted on new machines—just as you'd have to do without continuous deployment.

As for data and schema changes, I do NOT recommend having multiple databases. Instead, try to keep all schema changes to row, column, and table adds without any existing row, column, or table changes or deletions. As I mention earlier, you'd need to do this even if you didn't use continuous deployment.

When you must make significant changes to the schema, the proven technique is to plan out two releases.

- ❑ In the first release, create a version that understands the old and new schemas and can handle both (no significant new functionality, just the ability to handle both schemas).
- ❑ In the second release, once the first stabilizes and is solid, ship a version that depends on the new schema. Now, if there are any problems, you've got a safe rollback.

How do I work this?

How can you use continuous deployment for integration testing, partner testing, stress testing, and certification? Let's run through those.

- **Integration testing** You deploy your new build to production but set the custom routing to direct traffic only from your engineering team to the build. (The default is no routing at all.) The rest of the world continues to see your last release. This technique is called "exposure control." Now your team can test against real production with real production data and real production load using a build not exposed to customers.

  Eric Aside You'll need good diagnostics to analyze any failures you see in production. That's true with or without continuous deployment.

- **Partner testing** Partners deploy their new builds to production but set the custom routing to direct traffic only from their engineering teams to their builds. The rest of the world sees no change. Now partners can test against your production services without anyone seeing their new work, including their competitors.
- **Stress testing** You deploy your new build to production and test it out. Once verified, you use exposure control to increase the live traffic to your new build by increments—first 1%, then 3%, then 10%, then 30%, then 100%. You monitor service health throughout the process. If your services ever show signs of trouble, you capture the data and route traffic back to your last release (instant rollback).
- **Certification** Partners deploy their new builds to production and test them. Once the builds are verified, partners use exposure control to direct the certification team

to their new builds. The certification team certifies their builds in production, before customers or competitors see their new work. Once the builds are certified, partners can choose when to direct live traffic to their new builds.

- **Beta bonus!** You deploy a beta build. Once it's verified, you use exposure control to direct beta users to the beta build.
- **Experimentation bonus!** You deploy a variation of your current build. Once it's verified, you use exposure control to direct half the live traffic to your current build and half to the new variation. You utilize the differences you see in usage patterns to improve your services.
- **Auto-rollback bonus!** After you direct all live traffic to your new build, you leave the previous release in place. You connect your health checks to the exposure control. Now if your health checks ever indicate trouble, your exposure control automatically and almost instantly redirects traffic back to your previous release—day or night.

We're not in Kansas anymore

Microsoft engineers learned a great deal from our move into enterprise software a decade ago. Unfortunately, those lessons have misdirected our recent service efforts, driving us to build out extraneous environments in the name of service quality.

Maintaining extraneous environments drains our bandwidth, power, and hardware budgets and dramatically burdens our engineers, without providing real quality assurance. This needs to stop, and thankfully it is stopping as teams adopt continuous deployment.

With continuous deployment, you get service quality without the added costs. You also bag a bunch of bonus benefits to help you improve your services and better serve your internal and external partners.

There was a time when software development was done without source control systems. Now such a notion is not only laughable, it's unconscionable. Continuous deployment provides a similar capability for services. Someday soon we'll look back and wonder how anyone ever worked without it.

Eric Aside Currently, Bing and the Ads Platform have the only production implementations of continuous deployment I'm aware of at Microsoft. Amazon has one of the best-known systems in the industry.

My team is currently building a very simple form of continuous deployment. It uses an on-machine IIS proxy to provide exposure control to multiple versions of the same roles on the same machine.

From the perspective of the engineering team, we still deploy the same roles to the same machines as we always have. The difference is that those machines now host multiple versions of the roles, with exposure control directing the traffic we want to the version we want. Sweet!

February 1, 2011: "Cycle time—The soothsayer of productivity"

Nothing infuriates me more than wasted time and wasted effort. I'm not talking about training, reorgs, moves, morale events, or vacations. Those at least have the potential to be valuable in your life. I'm talking about build time, integration time, unused specs, incomplete features, blocking issues, excessive and persistent bugs, and over-engineered code and processes. You know—hours and days you'll never get back.

I broke down all this real waste years ago in my column "Lean: More than good pastrami" (earlier in this chapter). While I provided examples and suggested solutions for individual areas, I didn't really map the path to a better life for your particular team. Every situation is different. You need a path that discovers your most harmful waste, drives your team to resolve it, and rewards your team with a visceral sense of relief.

In manufacturing, the secret path to success is reducing inventory. Inventory hides your manufacturing issues. As you reduce inventory, problems appear. You fix the issues and then reduce inventory further. Gradually, your waste is eradicated, and your efficiency soars. In software development, the secret path to success is reducing cycle time. The shorter you make the time between concept and completion, the more roadblocks you face that have little to do with actual engineering. Fixing those problems unleashes productivity. Let me show you the way to free your engineering soul.

Eric Aside Many readers missed the point of this column, thinking it applied only to websites and web services. While a website was my example, shortening cycle time applies equally well to packaged products like Microsoft Office. The one difference is that for websites and services that release monthly or more often, a cycle is the time between releases. For packaged products or online services that release annually or less often, a cycle is the time to complete a feature from start to finish.

Many engineers question why techniques like reduced cycle time, co-location, early bug correction, and other lean concepts make a difference. To me this is surprising, because these same engineers don't question why optimizing inner loops, avoiding disk I/O, and trapping faults at their source all help software performance. If you don't see the connection you really should get out more.

What's done is done

The first step to shortening your development cycle time is determining how long your cycle takes. For services, it's the time between releases. However, for packaged products, shortening time between releases often can't be supported by the market. Thus, a better definition of a cycle is the time between starting detailed specification of a feature and having that feature completed. What does it mean to complete a feature? There's the rub.

You've got to define "done" for your features and for release of your services. Here are the definitions my team uses. We insist that the first four be done for every feature and the second four for every release. We release the Xbox.com sites every four weeks—and we LOVE IT!

"Done" for every feature:

1. All updated designs and code are reviewed
2. All automated tests written and passed
3. No ship-stopping bugs
4. All monitoring and health checks in place (feedback tools for packaged products)

"Done" for every release:

1. All localization and world readiness completed
2. Full test pass completed successfully
3. All quality areas signed off and partners signed off
4. All necessary release documentation completed

As you attempt to shorten the time between starting and finishing work, it's these eight "done" criteria that expose issues. Let's briefly discuss the common problems that arise and how to respond.

If you build it

This first requirement for "done," reviewing updated designs and code, only saves time, so let's talk about automated tests—unit tests, component tests, stress tests, acceptance tests, system tests, fault injection tests, and so on. Developers and testers should share in writing these tests. Who writes which tests varies by team. As they attempt to shorten cycle time, most teams struggle with their test harnesses and the time it takes to run the tests.

When it comes to reducing cycle time, you've got to distinguish between tests that run quickly and often and tests that run slowly and infrequently. Any tests that fall in the middle need to pick a side and be rewritten or refactored.

While it's nice to have one test harness, you can get away with two—one for fast and reliable check-in tests and one for full test passes. If you've got such a big team that even quick check-in tests take more than 10 to 20 minutes, then you've probably got a large enough team to invest in test prioritization and parallelization technology.

Likewise, if you've got such a large codebase that it takes more than 10 to 20 minutes to rebuild, then you've probably got a large enough team to invest in a highly parallelized build lab and build dependency logic. Remember, build, test, and check-in form software development's inner loop. Anything done to speed up your development inner loop creates a huge multiplier to overall productivity.

As for code branches, you never want to be more than one branch deep from the main branch. Integration is expensive, and each branch level adds another integration layer. Think about it. Say you were building personalized laptops. Having the distinctive components go through customs would crush your delivery schedule. Every branch level is like another customs station between your fixes and features and the main branch.

Roaches check in, but they don’t check out!

Cleaning up a large bug backlog before release can really slow down cycle time. Bugs take progressively longer to fix the longer you wait. Design and code reviews plus automated testing will help (as will refactoring spaghetti code and switching to test-driven development). Regardless, what really matters is finding and fixing bugs early. Short cycle times demand immediate bug fixing.

No matter what you do, you’ll still have bugs—we are human. Some of those bugs will be very difficult to find and fix, which will slow you down. The good news is that an architecture that is resilient to failure can alleviate the need to fix the toughest bugs—the intermittent ones. Resilient architectures allow you to collect data on these stubborn, sporadic slip-ups and fix them once they are finally understood.

Eric Aside I wrote more about resiliency in one of my more controversial columns, “Crash dummies: Resilience” (see Chapter 5).

How am I doing?

Monitoring and health checks are often treated as afterthoughts. This amateur-hour action increases the time needed to track down customer issues, which lengthens cycle time. This is just as true with designing and implementing customer feedback tools for packaged products.

Monitoring and health checks need to become forethoughts, designed in from the start. Consider why you are building your feature, and ask how you’ll know if it is performing as envisioned. That will tell you exactly how to monitor its use and inquire about its health.

All this data and your quick cycle times enable fast feedback loops and constant improvement. Be sure to spend time during every cycle reflecting on what you can do better in your product and with your engineering team. My feature teams and leads do this twice every release (every two weeks).

Eric Aside Making monitoring and health checks a forethought is my team’s most recent addition to the “done” list. Poor monitoring and insufficient health checks caused us to stumble in the fall, while we watched a partner team of ours shine in the same area.

Sign me up

We already talked about automation for full test passes, and localization processes are quite refined and fast at Microsoft, so the next area that typically causes trouble is sign-off. Quality areas (security, privacy, and so on) should be addressed and bugs fixed by all engineers as part of normal feature work. However, sign-off on these areas, as well as partner sign-off, can really slow down cycle time.

Even though quality is the responsibility of every engineer, sign-off works best if one engineer is assigned to shepherd each quality area through its process. Those engineers become the team specialists in their areas, a nice career opportunity for them that provides cross-group scope.

Since team specialists deal with their quality areas all the time, sign-off requirements and activities are far faster and easier for them than for other team members. In addition, team specialists develop relationships across the team and with corporate specialists in their area, which also speeds the process and provides growth for the entire team.

Eric Aside Another thing we did in Xbox.com to reduce cycle time is co-locate feature teams (including contingent staff and vendors where applicable). That gave us a 20+% increase in productivity (as measured by the size and number of features completed over a release).

How about a few more details

Using many of the techniques I've described, my team has managed to reduce the cycle time for our production releases from a few times a year to every four weeks. It's fantastic! Before I get into all the advantages we are seeing, let me cover two topics people often question.

How do you develop features or architecture changes that take longer than four weeks? There are two basic approaches: horizontal and vertical.

- The horizontal approach is to work on the large change a layer of the stack at a time. For example, first make the schema change, then ship the new service, then write the new model, then the new controller, and finally the new view. Each layer can ship within a four-week cycle.
- The vertical approach is to break the large change into smaller slices of functionality. You then complete each vertical slice end-to-end within a four-week cycle. If the slices lead to a disjointed user experience, you hide the slices from users until enough of them are complete.
- People often use a combination of horizontal and vertical techniques. Unfortunately, the horizontal approach often leads to over engineering of layers and hampers iterative

feedback. I much prefer the vertical technique, using the horizontal approach only as a last resort.

How do you handle sustained engineering? Sustained engineering fixes usually take about a month from identification through testing and release. Since we ship every four weeks, sustained engineering is just part of our regular work. **There are no sustained engineering releases except in the most unusual of cases, and more importantly, there is no special sustained engineering team.** We are in it together—we all feel the pain of our mistakes and the joy of our advances.

Life is good

Now that we've been releasing every four weeks for the last six months, we're really feeling the benefits.

- **Much of the overhead that engineers complain about is gone.** We had to remove it to succeed.
- **Slipping is manageable.** If a feature misses a release by a week or two, it still goes out within a month.
- **Releases aren't scary or crazy anymore.** We do them all the time, and you can cause only so much trouble in four weeks.
- **Our team gets more done in less time.** We're faster because of the streamlining that frequent releases demand.
- **We serve our customers well, and they notice.** Our dramatically improved response times to issues and feedback is greatly appreciated by our customers.

While there is pain involved in any change, shortening cycle time provides the immediate gratification of less overhead and quicker results. The team loves it, and I love it.

In the future, we want to be able to ship in one or two days, like some of our competitors (who are probably laughing at my team's long, four-week cycles). We don't plan to release that quickly all the time, but being able to do so will mean being even more streamlined. Once you start down this path, you get hooked on having so little between you and your customers, and that is a great place for everyone.

Eric Aside Why did we move the Xbox.com team to four-week cycles? Because several members of the leadership, including me, knew it was the best way to improve our team's productivity and customer quality. Trimming cycle time and work in progress is an old technique from Lean Manufacturing, which dates back to the 1930s.

Chapter 3

Inefficiency Eradicated

In this chapter:

As I described in "Lean: More than good pastrami" in Chapter 2, waste and evil are close companions in the work environment. Nowhere is that more evident than in group communications, a popular target of these columns, or in the proper use of unstructured time between projects. These areas affect whole teams, not just individuals, so their impact is multiplied.

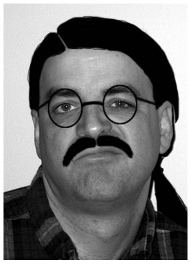
Specification documents (specs) and meetings hold a special place of honor in my museum of horrors. I guess that's because engineers spend so much time in meetings, often talking about specs. While I'd love to simply banish both from the world as we know it, meetings and specs do serve a purpose. The trick is to focus on that purpose and slice away all the excess.

In this chapter, I. M. Wright describes strategies for eliminating common inefficiencies. The first column deals with last-minute spec changes. The second tackles appropriate use of slack time in between projects. The third focuses on minimizing meeting malaise. The fourth tries to eliminate specs entirely. The fifth attempts to at least make specs shorter and simpler. The sixth solves distributed development. The seventh demonstrates how to properly optimize group work. The eighth demonstrates how to use checklists and single-piece flow to improve your processes. The last column advocates for decision making even under ambiguous circumstances.

Other columns have plenty more to say about group communications—everything from cross-team negotiation to dealing with nontechnical folks. Still others talk about actions individuals can take to improve their lot. But this set strikes at the core of what groups can do to make the best use of their limited time.

—Eric

July 1, 2001: "Late specs: Fact of life or genetic defect?"



You've hit code complete, you're burning the bugs, when what arrives in your Inbox? Look, oh joy, it's a new spec! Punt it, right? But wait, it's a key feature that you figured was spec-less, or as we often like to say, "The code is the spec."

> **Eric Aside** A feature is code complete when the developer believes all the code necessary to implement the feature has been checked into source control. Often this is a judgment call, but on better teams it's actually measured based on quality criteria (at which point it's often called "feature complete").

Of course, test is now furious because they didn't get the spec earlier and feel "out of the loop," it's too late, the code doesn't match, and they haven't tested it. Dev is upset because feature work was supposed to be finished, test is now mad at them for coding the "wrong" thing, there's a ton of rework to do, and what's worse, dev has been caught coding an improperly documented feature. It gets even more pleasant as people argue over the new spec, find holes, make changes, and basically churn the code to death at the very time it should be stabilizing.

For every change, churn, churn, churn

An extreme example, perhaps, but it's happened, probably more than once. Even if they aren't that late, specs often are incomplete or aren't reviewed and inspected in time for dev to start work.

So what happens? Churn, and lots of it. Dev starts coding too early. The spec has issues, so the code has issues. Someone points these out, ad hoc meetings are held, someone gets left out, the code is reworked, whoever was left out finds something else wrong, there are more ad hoc meetings, and so it goes.

What can be done about this? Some folks might say, "PMs are scum, persecute them till they produce." Even for me, that seems harsh. Specs come in late; it's a fact of life. The question is how you deal with it. I've seen a few different approaches.

Eric Aside I know eXtreme Programming buffs out there are yelling, "Get a room!" (a team room). I make the same argument in a later column, "Stop writing specs, co-located feature crews." However, Microsoft is a fairly diverse environment. Not every team can co-locate, and dependencies often make documentation a must, so we need more than one solution.

Hallway meetings

The first approach is the hallway meeting. A dev finds holes in the currently available spec and sees a PM passing by. A hallway meeting commences; some issues are worked out. The dev goes happily back to her desk thinking she now knows the right thing to do. The PM goes back to his office thinking the code will reflect what he wanted. Maybe they are thinking the same thing, maybe not. Maybe test and ops would agree with the solution, maybe not. Maybe they thought of everything, maybe they didn't. Maybe this is the best way to handle changes, maybe monkeys will fly out of my… well, you get the idea.

Committee meetings

A second approach is the committee meeting. It goes by other names on other teams, but it's basically a leads' meeting to discuss spec changes. Often they're held on a regular basis, and the group of leads gets together to talk about holes or problem areas in the specs and work out solutions as a group. The lead PM writes up the results and mails them out to the whole team.

The good news: committee meetings include the right folks, come to final decisions, and then document and communicate those decisions to the team. The bad news: committee meetings are a frigging nightmare. They are long, painful, and exhausting. They use up huge cycles of critical resources. They block progress and form the worst kind of bottleneck—self-inflicted and self-perpetuating.

Spec change requests

The approach I like most is the spec change request (SCR), also known as a design change request (DCR) with a twist. It's a combination of the committee meeting and hallway meeting with a few key differences. You start with an idea of how you'd like to change or add to a spec. Maybe you arrived at the idea on your own, maybe through a hallway conversation, maybe through a leads' meeting.

Regardless of whether you're the PM, dev, test, or ops, you write up the idea in an e-mail with the subject line "SCR: <*affected spec*> - <*short description of change*>." You end the e-mail with these words in bold, "**Unless there are strong objections, this is now considered spec.**" Then you send it to the PM, dev, test, and ops folks who are most directly affected by the change. A few days later, after adding whatever alterations are suggested by peers, you send it to the rest of the team and track it with other SCRs in RAID and/or a public folder.

The key is that the change is documented and reviewed but does not block progress. Objections are almost always the exception, not the rule. The dev can proceed whenever she likes, trading risk of objections against time. Typically, a dev waits till the SCR is sent to the full team after the initial alterations.

Prevention is the best cure

Of course, the best thing is for a spec not to be late in the first place or at least for it not to blindside you. That's where T-I-M-E Charting can help. In T-I-M-E Charting, the first spec lays out the design of the entire project. Not simply a requirements document, not a set of mini-specs, but a high-level spec of the project much like a high-level architecture document a dev lead might write. It should lay out what functions and UI the project will have and how they will act together, leaving details for later specs. All future specs and features should be referred to by the first high-level spec.

Now dev, test, and ops can make plans that account for all future features. They can make a better integrated product that feels smoother to the user. PMs can also use the first spec to schedule the rest of the specs, hitting the high-priority ones first, without worrying about missing something or surprising someone. It's an idea whose T-I-M-E has come (couldn't resist).

Eric Aside Totally Inclusive Mutually Exclusive (T-I-M-E) Charting, from Donald Wood, never quite caught on in the form that a peer of mine, Rick Andrews, originally envisioned it. However, value propositions, vision documents, cross-product scenarios, and thoughtfully designed prototypes now serve the same purpose.

June 1, 2002: "Idle hands"

Your dev team hit zero bug bounce (ZBB) two weeks ago, and suddenly you realize—you've hit a lull. Any dev who has hit ZBB on a box product knows about the lull. If your team is on Internet time, feel free to stop reading now. (Wait a minute, where did you find the time to read that first sentence? Get back to work!)

Eric Aside Zero bug bounce (ZBB) describes the first moment in a project when all features are complete and every work item is resolved. This moment rarely lasts. Often within an hour a new issue arises through extended system testing, and the team goes back to work. Nevertheless, ZBB means the end is predictably within sight.

BTW, my team now works on Internet time, and we still manage to have monthly morale events and do our share of reading and innovating. Our secret—lean and agile baby! Read about it in Chapter 2.

ZBB marks the team shift from being blocked by dev to being blocked by test. (Being blocked by PM has no transition.) After handling the initial wave of new bugs the first couple of weeks after shipping, most dev teams enter "hurry up and wait" mode, pouncing on new bugs when they arrive and otherwise just wondering what to do.

The crazy, scary part is that the lull can sometimes last from ZBB until the first milestone of the next version. That could be months on big projects! A dev manager's hands are always full, so it's easy to forget that two-thirds of the team members are idle, and you know what they say about idle hands—well, it's not good.

Baby did a bad bad thing

Here are a few very bad things that idle devs often do:

- **Poach bugs.** After ZBB, your team should be in lockdown, which means that all bugs go through triage before a fix is even considered. Idle devs sometimes sit by their desks hitting F5 on RAID (now Product Studio) waiting for a bug to appear. When they don't see one, they check the active bugs headed for triage, find a juicy one, and start digging. Before you know it, they've got a fix and are looking to sneak it in. That's poaching, and no self-respecting dev should do it.

  Eric Aside In software engineering, bugs have traditionally meant mistakes in the code. However, internally we use the term "bug" to refer to anything we want to add, delete, or change about the product. Externally, people generally call these "work items," some of which may be code mistakes. I prefer the term "work item" so that I know which "bugs" are really bugs.

  Who knows if the triage team will accept that bug? Who knows if the dev fixed the right bug, as opposed to a larger or smaller related bug? Investigate potential show-stoppers—sure. Poach—never.

- **Fix bugs that are not logged.** Okay, a bug made it through triage and you are fixing it. You notice other bugs nearby, often related to the original. No one has logged them yet, but what the heck. There you are in the code; the bugs are right in front of you.

Why not take care of them now, when the getting is good? Ahhhhhhhhhh!!!!! Stop right there!

Your team performs code reviews to prevent this evil nonsense. In these days of trustworthy computing, the team should code review every check-in throughout the project. During lockdown, you should code review every change with three sets of eyes (the dev and two other people). As for the other bugs that you find—log, triage, and track them.

> **Eric Aside** There's a great Calvin and Hobbes cartoon in which Calvin magnanimously opens the front door to let out a fly, only to allow three more back inside in the process. That's why you study and triage every bug toward the end of a project. Once my team changed the value of a single parameter a month before we released. A week later testers across the company noticed that all applications froze whenever you opened the CD tray. Eventually, we traced it back to the seemingly innocuous parameter, and reverted the change. You just never know.

- **Fix postponed bugs.** Naturally, you shouldn't be fixing postponed bugs before RTM, but should you fix them while planning the next version? Uh, no. During the project, the team judges which bugs will have the most impact on our customers and must be fixed—but you have no way of verifying that these were the correct choices until you ship. After release, you no longer have to guess. Product Support Services (PSS), Watson, and Microsoft Consulting Services (MCS) will tell you. Candidly. Use the post-poned bugs as a reference to understand why these bugs weren't fixed originally. But don't second-guess your real customers. Go to the source to fix the bugs that are really affecting your users.
- **Rewrite "yucky" code.** Devs hate "yucky" code. It's embarrassing, unreadable, and unmaintainable. So when they have some extra time on their hands, devs will often say to themselves, "Gee, I don't have a spec, so I can't write anything new. How about I rewrite that yucky code I hate instead." They know that they could do better given a second chance, and they will. Devs will write much better code the second time, and more clearly, with far fewer bugs than the first time they wrote it.

  Unfortunately, the rewrite will actually have more bugs than the current yucky code has today because of all the months, even years, of testing and fixes that the yucky code received after it was first written.

  Sometimes a rewrite is necessary to make the code more performant, scalable, reliable, secure, or adaptable to new technology. In that case, make the rewrite a feature, and then spec and schedule it like you would any other feature. Otherwise, don't be a fool and regress a ton of nasty bugs while adding no value to your customers.

Eric Aside This goes for refactoring too, as much as I hate to say it. Even if the refactoring is computer generated, you just never know. This doesn't mean you shouldn't refactor or rewrite code, regularly. It means you shouldn't do so arbitrarily. Decide and commit to it as a team, be sure sufficient unit tests are in place to minimize the introduction of new bugs, and do it right.

- **Wage wars over coding style.** Talk about the ultimate dev team time suck—arguing over white space, braces, and Hungarian have to be in the top five. Keep this in mind: using a consistent coding style has great benefits to the maintainability and quality of your code base, but the specific style your team chooses makes little difference. You are the dev manager; pick one and go with it. Who said this was a democracy?

Tell me what I must do

Enough of the dark side of idle time. What can your dev team focus on that's constructive during quiet times?

Naturally, your test team will insist that devs find bugs during the time before RTM, but most devs are terrible at finding bugs, even in someone else's code. Your PM team will insist that devs spend all their time reading and reviewing specs after RTM, but that won't keep a dev team happy, engaged, and motivated.

So what can a dev do during the lull? Here are a few ideas:

- **Analyze your bugs.** Look for patterns in the bugs that your team fixed during the past product cycle. What were the common mistakes for individuals and for the team? What can each member of the team focus on next time to produce a better product?
- **Write tools for your group.** While devs aren't often great at finding bugs, they are pretty terrific at writing tools to help find bugs. They can also write tools to smooth out processes, like check-in, setup, build, and prop. Instrumenting code or writing a good harness can go a long way toward promoting good feelings with the test team. Naturally, you should check the Toolbox website first to see if a tool that meets your needs already exists.
- **Make your PM happy by working on prototypes of design ideas.** Writing prototypes is great; just don't write them in your usual code base. Try a different language or at least a separate build. The big mistake with making prototypes within the normal code base is that PMs and upper managers start thinking that the code is almost ready to ship, when in fact there are often all kinds of issues with localizing, platform dependence, logo issues, roaming, performance, security, and compatibility. Confusing prototypes with shipping code can mess up schedules as well as expectations. In contrast, writing a prototype in another language can be a great learning experience. Speaking of which…

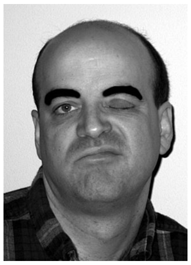
Eric Aside It's been said before but bears repeating, "Don't ship prototype code." It doesn't save time, it costs time. Just don't do it. Prototypes are for learning—that's all. In addition to writing prototypes in another language, I used to hook the escape key to an abort call. That way, if my boss ever got too excited watching a demo, I'd hit the escape key, watch it crash, and then point out, "Of course, it's not exactly ready to ship." For more on prototyping see "My experiment worked! (Prototyping)" in Chapter 6.

- **Learn new technologies or skills.** Folks always complain that they don't have enough time to learn new technologies or skills and that they can't get the training they need to move up. Well, the quiet times are perfect for this. Don't let the opportunity pass you by.
- **Talk to research.** Right after ZBB is the perfect time to talk to the research team. It's early enough to adopt some new technology and quiet enough to learn about it and figure out what you can use. By the time you ship and begin planning the next release, you could have a prototype ready with all the risks resolved and really wow your team. In addition, you and your research contact can plan new areas of research work that will be ready for future products. This is so valuable and easy to do.
- **Write a patent disclosure or white paper.** When else do you have the time to reflect and write about what you've done? If a dev on your team has come up with a novel idea that added a nice or significant touch to your product, then have your dev write a patent disclosure. It's easy, short, and a huge morale boost. Go to the Patent Group home page for more details. If you want to document information or share an idea with other teams, write a white paper. It's relatively easy to do and can bring respect and influence to the author and your team.
- **Reflect on your career.** Last but not least, these quiet dev times are ideal for examining your career status. Are you where you want to be? Is your career moving in the right direction? Are you ready for a new challenge? What do you need to do to stay engaged and motivated? If upon reflection you feel that you need to make a change, the earlier you put the wheels in motion, the better off you'll be.

Waste not, want not

Far too frequently, time spent between versions is wasted needlessly, often on tasks that harm instead of help. With just a little thought and consideration, your dev team can be improving themselves, the product, their outlook, and the entire group without getting into any mischief. Don't pass up this opportunity for you and your team. Plan for your downtime and keep the momentum moving forward.

June 1, 2004: "The day we met"



Quit wasting my time. Would you, could you, PLEASE quit wasting my time? Maybe if I jump across the table and duct tape your mouth shut, I could take action instead of sitting here incredulously while you incinerate 60 minutes of my life. How does calling a meeting give people license to act like you're worthless? If time is money, most meetings are a market collapse. I am so tired of people who could sooner drive a bus off a cliff than run a decent meeting.

Well I'm not going to take it anymore. If you force me into a meeting room, be prepared for me to call you on any stunts you try to pull. You waste my time, and I'll ensure yours gets torched with it. Don't like it? Don't test me. What am I going to call you on? Listen up, 'cause here it comes…

Why are we here?

The first question I'll interrupt your self-serving soirée with is, "Why are we here?" What was the point of us getting together? Was there a reason? If you haven't made that reason clear to everyone, we all probably think it's something different and will chase our tails for the allotted time, accomplishing nothing. I don't know—maybe send an agenda and the documents that we're going to discuss in advance? Thanks.

If you did make the point of the meeting clear, I'm probably reminding you to stick to the point! I don't care if there are 50 other meetings with 50 other decisions to make, topics to cover, or bits of information to share. I'm in this meeting now, and I darn well want to at least bury this one. If someone wants to talk about something else, let him put on his own show after we're done.

Eric Aside How do you politely cut off someone trying to switch topics? My favorite approach is to say, "Let's get closure on this topic first, then we'll focus on your topic." Typically, after you close on the first topic everyone will want to leave. The interrupter will have to schedule a separate meeting with the right people (much better). Should the interrupter insist that closure on his topic is necessary first, discuss why that is the case (which actually focuses on the original topic). If the interrupter is right, your meeting is premature and should be rescheduled. No one will mind leaving.

What are we trying to do?

My next question will be, "What are we trying to do?"

- **Are we trying to reach a decision?** Great, let's decide and skip the idea generation, status checks, and rumor mill.

- **Are we trying to share information (like a status meeting)?** Great, then get through the information list and stop trying to make decisions or solve problems.
- **Are we trying to generate ideas?** Great, then capture everyone's ideas and stop critiquing or judging what's possible. At the end, pick the best idea and be done with it.

The point is that combination meetings are ineffective and wasteful. Know why you are all there and what you are trying to accomplish. If you need to switch contexts, be deliberate about it, and let everyone know that the rules have changed. Otherwise, you'll waste everyone's time, spin endlessly, and eventually have to meet again. When you do, don't bother inviting me; I'm not coming.

Eric Aside A common special case of this issue is bringing up design issues at Scrum meetings. Scrum meetings are about sharing information, not generating ideas or making decisions. Nothing derails a Scrum meeting like a design discussion. However, because design discussions are worthwhile, we keep a list of discussion topics on the whiteboard during the scrum. When the Scrum meeting is complete, whoever wants to can stay and participate in the design meeting.

Why are they here?

Okay, so we've got a reason for meeting and we know what we're doing. Now why are *they* here? You know—the people who don't belong here. The people who are asking the unnecessary questions, who are repeating other people's points, who have to speak up just to say they agree. Why are those people here?

The length of a meeting is directly proportional to the number attending, and I doubt that the relationship is linear. You should invite only those who NEED to be there.

- **Trying to reach a decision?** Invite the decision makers. Everyone else can find out later via e-mail. All the necessary decision makers can't attend? Cancel the meeting. NOW! Otherwise, you'll have to recap the whole meeting again when everyone can attend.
- **Status meeting?** Invite the people who will share their status. Everyone else can find out later via e-mail. Some status people can't make it? I guess they're slackers.
- **Brainstorming meeting?** Invite a few creative, open-minded people who'll make the meeting successful. Everyone else can find out later via e-mail.

Sometimes you must invite a few others who are key to the meeting's success: facilitators, mediators, cheerleaders. But that's it. If too many others are signed up to attend, cancel the meeting. (You can tell how many people plan to attend because when folks accept a forwarded meeting, you receive the confirmation.)

Try booking a small room; it dissuades uninvited guests. Try scheduling the meeting for just 30 minutes—it makes folks show up on time and keeps the meeting moving. You can say it's a "working meeting" and even use information rights management (IRM) to prevent forwarding the appointment if necessary.

Why am I hearing this now?

For important topics, you don't want to surprise key players. No one likes to be rushed in making critical decisions, and no one wants to be uninformed about critical areas. If you need the meeting to go smoothly, talk to the key players beforehand. You can discover the issues, negotiate a compromise, and get everyone on the same page in advance. Then the meeting becomes a mere formality. This is a good practice for every decision meeting, but it is time-consuming. For critical decisions, it is a critical step.

What are the next steps?

So the meeting is done, finished, kaput, right? Wrong! Meetings are like Hollywood horror show zombies. They come back to life and eat those who remain. Determine the next steps, and document them in e-mail. That is the way to make dead meetings stay dead.

Address the e-mail to all attendees and cc: everyone affected by the outcome. Include a short meeting summary of the decisions made, information shared, or ideas generated. Then list the next steps specifying who does what, when. Now, finally you can move on in safety.

See, it's not so hard to respect people's time. Meetings are costly in so many ways. Of course, they are necessary for strong group communications. But if you run them, run them well. Everyone will appreciate it, and you'll get more done.

July 1, 2006: "Stop writing specs, co-located feature crews"

I'm not a Program Manager (PM). I've never been a PM. I'm not likely to ever become a PM. It's nothing personal against program managers. I've known great ones I count among my friends. I certainly have no right to tell PMs how to do their jobs.

That said, PMs should stop writing specs. Period. They are wasting my time; they are wasting my group's time; and they are wasting the company's time. You can almost hear the sound of quiet residual crunching as spec termites chew away at company and customer value. It makes me nauseous.

It's not just PMs though. Developers need to stop writing dev specs. Testers need to stop writing test specs. The madness must stop. The waste must stop. We must regain our senses and take back our productivity, along with our sanity.

Eric Aside This column was easily among my most contentious in terms of response. As you can see in the next paragraph, I guessed it would be. The biggest misunderstanding about my message was the difference between formal and informal documentation. I argue that co-located, cross-discipline teams need only informal documentation, like photos of whiteboards stored on a wiki with some minimal commentary. Teams divided by distance or discipline need formal documentation, like detailed specifications.

Have you lost your mind?

"Surely you can't be serious?" I hear my conscientious readers say. "You've been preaching quality (see "Where's the beef" in Chapter 5) and design (see "Resolved by design" in Chapter 6) for years. You've been telling devs not to act before they have the spec and not to code before they've thought through the design. Are you saying you were misguided, or even, perhaps, wrong?" No, of course not.

Feature teams must understand the user experience before they create it, and devs must understand the internal design enough to explain it with a straight face to peers before they implement it. But neither of those steps requires formal written documentation.

Why do we have formal written specs? Customers don't need them. Marketing and product planning groups don't need them. Even content publishers and product support get limited use from specs. So who needs these wanton wonders of waste? To find out, throw specs away and see who screams.

Therein lies a dilemma

If we no longer had specs, devs and testers would cry, "How am I supposed to know what the code should do?" Tell them to discuss it with PMs and they'd holler, "PMs don't just hang around my office all day. I need specs written down. I need to review them, change them, and update them."

Ah yes, there's the rub. Not that devs and testers need to review, change, and update specs, but that PMs don't hang around to discuss the user experience, implementation, and test strategy all day. Well, what if they did?

What if PMs stayed all day in the same open area surrounded by whiteboards with devs and testers who were working on the same feature set? Would we still need formal written specs? Wait, I hear more screaming.

Special needs

Without formal written specs, teams that depend on features from other teams would protest, "How are we supposed to use your code if we don't know how it works?" Good point. If PMs are hanging around the feature team all day, they can't also be on call with all the downstream teams, and we can't fit everyone into the same team room. However, downstream teams don't need a spec—they need a mini-SDK, which component teams should be providing anyway, and which adds great value.

Without formal written specs, the compliance police would bark, "Where's the <insert your favorite bureaucratic suppository here> document?" Another good point. The compliance police keep us out of harm's way. It's an important if thankless job, and they often require formal written documentation to do it. However, the compliance police don't need a spec either. They need complete compliance documentation, which often has different information in a different form than a spec.

Eric Aside Who are these "compliance police"? They are regular engineers who focus on key areas Microsoft must ensure are correct in our products, such as security, privacy, world readiness (no inappropriate euphemisms or references), and compliance with all applicable laws and regulations. Examples of typical documentation they require include threat models (security), privacy statements, and licensing terms.

In both cases, you don't need formal written specs. Instead, you need specialized documentation that is easier to write because it's not open-ended.

I don't recall

So do we still need formal written specs? I can't remember all the cases, so let's recap:

- PMs spend their days in the team room discussing the user experience, implementation, and test strategy with the feature team.
- The team writes mini-SDKs for downstream teams.
- The team fills out required compliance documentation.

It's good that I wrote that down. Oh wait, that's a problem.

People are forgetful. You've got to write ideas down, particularly when you are constantly switching between projects. Naturally, if a feature takes months from start to finish you might even have people leave the team and lose the information entirely.

Stick to one thing

But what if you worked on only one feature at a time? Then it wouldn't take as long, and you wouldn't be switching between projects. There would be little chance of anyone leaving, and remembering ideas would be much easier. You'd just need to capture whatever was on the whiteboards with a digital camera and paste it into a wiki, Word document, or OneNote notebook.

It's like having specs, only without the mind-numbing tedium. That leaves you more time to think and collaborate at the whiteboard, and less time at your desk pushing pixels and words around.

Okay, you keep the feature team in close quarters with lots of whiteboards. You work on one feature at a time till it's done, documenting your decisions with a camera. You write specially targeted documents that add value downstream. This sounds like Lean software development (which you can read more about in the article "Lean: More than good pastrami" in Chapter 2). Bingo! That's what you get when you cut out the waste.

You ready?

Very few teams could stop writing formal specs tomorrow. They haven't adopted the feature crew concept of working on one feature at a time from start to finish, and they aren't co-located in a team room with whiteboards.

However, that's starting to change. Groups are co-locating because it's faster and easier to get work done. Groups are forming feature crews because you get higher quality faster and leave behind less incomplete work. Take those trends, put them together, and you can kiss specs goodbye forever. It's more than a dream; it's a homecoming to simpler days, but with the wisdom gained by years of hard-fought experience.

Eric Aside My first tricky project as development manager for the Xbox.com team was to co-locate the six Scrum teams, including removing the cubical walls within each team room. Those six teams had been operating on the same big release for Kinect and Windows Phone for several months before the office shuffle, and several months after the shuffle. The timing and all the team velocity data that Scrum provides made this a great opportunity to measure the impact of co-location. The four-week average velocity for five of the six teams increased between 20% and 63% (the sixth team was releasing a beta and halted new development). That 20% increase is like getting an extra day a week of productivity and still having a two-day weekend. The team that increased 63% (that's three extra days a week!) also decreased the size of their stories, left stories unassigned so that the first available person could pick them up, and increased cross-discipline pairing for stories with many unknowns. The only documenting the teams did was maintaining OneNote notebooks and completing the essential compliance work.

February 1, 2007: "Bad specs: Who is to blame?"

Specs, by and large, are terrible. Not only PM specs, but dev and test specs too. By terrible, I mean difficult to write, difficult to use, and difficult to maintain. You know, terrible. They are also incomplete, poorly organized, and inadequately reviewed. They've always been this way and they aren't getting better.

I'd love to blame PMs for this, partly because I enjoy it, but mostly because they are the leading source of awful specs. However, the facts don't support blaming PMs. Everyone writes bad specs, not just PMs. Even PMs who occasionally write good specs, mostly write poor ones. And good specs are still difficult to write and maintain, regardless of who authors them.

If PMs aren't to blame for shoddy specs, who is? Management would be an easy target—another group I'd enjoy blaming. It's true that some organizations, like the Office division, traditionally produce better specs than others. So clearly management has a role. However, Office has changed management many times over the years, so the cause must be deeper than the people in charge.

It's a setup

It's clear that the blame falls squarely on the spec process—how we write specs and the tools we use to write them. The process is cumbersome, difficult, and tedious. The templates are long, intimidating, and complex to the point of being intractable. Basically, we've made writing good specs as hopeless as winning a marathon in a fur coat and flip-flops.

Anal anachronistic alarmists will say, "The spec process is absurdly dysfunctional for a reason. All template elements and process steps are needed to avoid past catastrophes." See, you never have to worry about too much bureaucracy from on high when there's plenty down low where it counts.

Dysfunctional processes always come from the best of intentions. The trouble is that the original goal and intent was lost somewhere along the way. Revive the goal and intent, and new and better ways to achieve it will present themselves.

Eric Aside I worked in Boeing research for five years. Not all, but most of the bureaucracy there seemed to come from the top. I've been at Microsoft for 16 years. Not all, but much of the bureaucracy here seems to come from the bottom. We are free to act independently at the lowest levels. Sometimes that means we're given enough rope to choke ourselves.

Communication breakdown

The goal of all PM, dev, and test specs is to communicate design and design decisions to people across time and location. We want to make that communication easy and robust, with plenty of feedback and quality checks.

In case you missed it, those were four separate requirements:

- Easy
- Robust
- Feedback
- Quality checks

Each requirement can be satisfied with a different solution. The approach, "We'll just add more sections to the spec to cover all requirements," is as idiotic as, "We'll just add more methods to the class to cover all requirements." Instead, let's take on the requirements one at a time.

Keep it simple and easy

The spec needs to be easy to write, understand, and maintain. It should use standard notation, like UML, for diagrams and common terminology for text. It shouldn't try to be too much or say too much.

The simpler the format the better. The generic spec template in the Engineering Excellence Handbook has 30 sections and three appendices. The superior Office spec template has 20 sections. Both are far too complex.

A spec needs to have three sections plus some metadata:

- **Requirements** Why does the feature exist? (Tied to scenarios and personas.)
- **Design** How does it work? (Pictures, animations, and diagrams are especially useful.)
- **Issues** What were the decision points, risks, and tradeoffs? (For example, dependencies.)
- **Metadata** Title, short description, author, feature team, priority, cost, and status.

That's it. The status metadata could be a workflow or checklist, but that's the limit of the complexity.

"But what about the threat model? What about the privacy statement? The instrumentation or the performance metrics?" I can hear you demanding. Get a grip on yourself. Those items are quality checks I'll talk about soon. The spec structure itself is simple, with no more or less than it needs. It's easy to write and easy to read.

Online Materials Spec template (Spec template.doc)

Make it robust

The spec needs to be robust. It must verifiably meet all the requirements, both functional requirements and quality requirements. "How?" you ask. What do you mean, "How?!?" How would you verify the requirements in the first place? You'd write a test, right? Well, that's how you write a robust spec. In the first section, when you list functional and quality requirements, you include the following:

Unique ID	Priority	Functional or quality	Short description	Related scenario(s)	Test(s) that verify the requirement has been met

If you can't specify a test to verify a requirement, then the requirement can't be met, so drop it. Can't drop it? Then rewrite the requirement till it's testable.

Eric Aside I believe there is a basic equivalence in solid designs between tests and requirements. Every requirement should have a test. Every test should stem from a requirement. This results in clear, verifiable requirements; more comprehensive tests; consistent completion criteria (all tests pass = all requirements met); and better designs because test-driven designs are naturally simpler, more cohesive, and more loosely coupled.

Get feedback

The more eyes that see a spec before it's implemented, the better it will be and the less rework it will require. You want feedback to be easy to get and easy to give. At the very least, put draft specs on SharePoint, using change tracking and version control. Even better, put drafts on a wiki or a whiteboard in the main area for the feature team.

How formal does your process, feedback, and change management need to be? As I discussed in a previous column ("Stop writing specs, co-located feature crews" earlier in this chapter), the degree of formality necessary depends on the bandwidth and immediacy of the communication. People working on the same feature at the same time in the same shared workspace can use very informal specs and processes. People working on different features at different times in different time zones must rely on highly formal specs and processes.

Regardless, you want the spec to be fluid till the team thinks it's ready. How will you know it's ready? It's ready when the spec passes inspection by the test team using the quality checks.

Check that quality is built in

Here is where our current specs go farthest off base. Instead of adding security, privacy, and a host of other issues as quality checks, groups add them as separate sections in the spec. This is a disaster, and here's why:

- Specs become bigger and far more complicated.
- Authors must duplicate information across sections.
- Bottom sections get little attention, causing serious quality gaps.
- Designs become incomprehensible because their description is spread across multiple sections.
- Mistakes and gaps are easy to miss because the whole picture doesn't exist in one place.
- Updates are nearly impossible because multiple sections are affected by the smallest change.

Instead, the quality checks that apply to every spec are kept in a list everyone can reference. The first few checks will be the same for every team:

✓ Are the requirements clear, complete, verifiable, and associated with valid scenarios?

✓ Does the design meet all requirements?

✓ Have all key design decisions been addressed and documented?

The next set of quality checks is also fairly basic:

✓ Have all terms been defined?

✓ Are security concerns addressed?

✓ Are privacy concerns met?

✓ Is the UI fully accessible?

✓ Is it ready for globalization and localization?

✓ Are response and performance expectations clear and measurable?

✓ Has instrumentation and programmability been specified?

✓ Are there issues with compatibility?

✓ Are failures and error handling addressed?

✓ Are setup and upgrade issues covered?

✓ Are maintenance issues addressed?

✓ Are backup and restore issues met?

✓ Is there sufficient documentation for support to do troubleshooting?

✓ Are there any potential issues that affect patching?

A team may also add quality checks for their product line or team that reflect particular quality issues they commonly face.

Online Materials Spec checklist (Spec checklist.doc)

The key is that the design section describes the feature completely, while the quality checks ensure nothing is missed. Yes, that means the "How" section could get pretty big to cover all the areas it needs. But those areas won't be rehashes of the feature specialized for each quality requirement (security for the dialog, privacy for the dialog, accessibility for the dialog).

Instead, the areas will be the feature's logical components (the API, the dialogs, the menus). Duplication is removed, each feature component is described as a whole, and all the quality requirements are incorporated into the design in context.

Eric Aside In an interesting and funny coincidence, the day after this column was published, Office simplified their spec template to a single design section and a published quality checklist. While I couldn't claim the credit for the change, I did feel vindicated.

What's the difference?

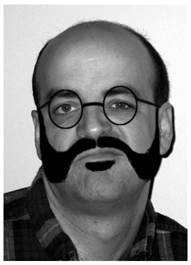
With all those checks and tests added, you might ask if I've simplified specs at all. Here are the big changes:

- The number of sections is reduced to three (Requirements, Design, and Issues).
- Designs are described completely in one section.
- All functional and quality requirements can be verified.

I've also talked about opportunities to make specs less formal and easier to understand.

Who's to blame for bad specs? We all are, but bad specs are mostly the result of bad habits and poor tools. By making a few small changes and using vastly simplified templates, we can improve our specs, our cross-group communication, and our cross-discipline relations. Altogether, that can make working at Microsoft far more productive and pleasant.

February 1, 2008: "So far away—Distributed development"



If you are a software geek, like me, being the product support technician for your friends and family comes with the territory. While it's painful to watch your family struggle with software, particularly if you helped write it, at least you can tell them, "Back off, I'm a computer scientist," and repair whatever is wrong. Sure, you'll cringe as you undo their failed "fixes," but in time you'll set things straight. That is, if you live nearby.

If your mom lives a thousand miles away, the call might sound more like this:

> Mom: *Honey, they changed my password and now my e-mail doesn't work.*
>
> Me: *Okay, log on to your computer and open Outlook.*
>
> Mom: *Using what password?*
>
> Me: *Use whatever password you normally use.*
>
> Mom: *Not the new e-mail one, just my old one.*
>
> Me: *Yes. Then open Outlook.*
>
> Mom: *Where do I type in the password?*
>
> Me: *On the main screen where you click on your name and then type a password.*
>
> Mom: *What main screen?*
>
> Me: *Wait, can you just open Outlook right now?*
>
> Mom: *Yes, I've got it open, but you told me to log on.*
>
> [Another hour like this moving through menus, dialog boxes, and buttons…]

This experience is far better if you can get a tool like Remote Assistance to work, but getting that up and running behind firewalls and routers can be equally entertaining. What should be a five-minute fix becomes an hour of acute aggravation. This order of magnitude multiplier for time and trouble comes into play any time you work across time and distance, which brings us to the topic of distributed development.

Doesn't anybody stay in one place anymore?

It's becoming far more common to run development for one project across global locations. While the added diversity and talent should be a huge advantage, the results are often frustration, delays, and disconnects in quality and functionality. Why? It’s due to the big, bad Bs—bandwidth, boundaries, and being there.

There’s insufficient bandwidth for clear communication and fast network access to central services. The boundaries between project work at the different locations are poorly defined, causing additional communication, conflicts, calamities, and cleanup. And because the different teams are separated, the one-on-ones, drop-ins, hallway conversations, and other daily human interactions you get from being there don’t occur.

With all this trouble you might wonder why we bother with distributed development. No, you fool, it’s not the money. Engineering salaries and costs are converging for many roles in China and India, and distributed development adds overhead. The real reasons for distributed development stem from the talent and the markets. The computer science talent pool in Brazil, Russia, India, and China is growing at nearly 20% per year and had 1.5 times the number of software engineers in all of the United States by 2010. They can’t all move to Redmond, and we wouldn’t want them to because their home markets are critical to our success.

So if you’re running a team, you’d better learn how to deal with the big, bad Bs. Let’s break them down.

I get so tired when I have to explain

The first big, bad B is bandwidth—bandwidth between people and between computers.

Clear communication between people and teams is critical to success, as I’ve said many times before. The amount of information that is communicated between two people in the same room far exceeds that between people over video teleconference (VTC), which in turn exceeds Live Meeting, which exceeds the telephone (as shown in the exchange with my mother), which exceeds IM, which exceeds e-mail. In other words, e-mail is incredibly lame as a communication vehicle, so naturally it’s the most popular.

Eric Aside Using the Microsoft Roundtable device with Live Meeting (now Lync) is now as good if not better than VTC. If you haven’t tried Roundtable yet, you are missing out on some cool technology. It automatically centers on whoever is talking in a natural, seamless manner.

Actually, people use e-mail because it’s convenient, asynchronous, and works across time zones. Unfortunately, because e-mail has such little bandwidth, the communication is poor and often several round trips are needed to gain comprehension and answers. Because of time zone differences, a round-trip often takes a day; thus e-mail communication proceeds at a glacial pace. Bandwidth between computers can also be quite slow, which means quick source control, file, or database operations in Redmond may take hours overseas.

How do you beat the big, bad bandwidth issue? There are only two ways—increase the bandwidth or cut down on the necessary communication.

- You can increase the bandwidth by using higher bandwidth communication tools, like VTC and Live Meeting. These work even over low network bandwidth lines. However, the tools are synchronous, so you must reserve overlapping work time for communication with distributed team members. That means if you're in Redmond and other team members are in China, you should reserve 4–5 P.M. Pacific Time every day for scheduled and impromptu meetings with your peers in China.

  Eric Aside Live Meeting has now been combined with Office Communicator in a new Microsoft product called Lync. It's really cool to have IM, video and audio messaging, IP phone, conference calling, and Live Meeting all in one Office-connected app. Yeah, I sound like a marketer, but it is pretty sweet.

- You can cut down on necessary communication by crafting far more careful e-mails. Mitigate questions and confusion by providing additional information and answering common questions in advance. It will take you a bit longer to write e-mail this way, but it won't take you days, which is how long the communication will take otherwise.
- You can cut out the need for a whole class of e-mails and phone calls by keeping a SharePoint wiki with project information. Fill it with how-to topics, discussion archives, checklists, documents, and project mail, schedule, and status. Any time a new team member arrives at any location, assign them to update the site with improved instructions or missing documentation.
- You can also cut down on necessary communication by creating clear project boundaries between the work that is happening at different locations. Isolate local communication, including caching of source control and databases. Then you need cross-group communication only when people or information cross boundaries. That brings me to the next big, bad B.

Doesn't help to know that you're just time away

The second big, bad B is boundaries—boundaries between work at different project locations.

If you've never worked on distributed projects or teams before, you might foolishly think that distributed groups are all just one happy team. It's this kind of naïve thinking that causes managers to allow individual team members to work alone from remote locations. Remember, being ignorant, naïve, and foolish is just one step from being stupid.

A project team residing in three separate sites is not one happy team; it's three potentially happy teams working on one potentially successful project. By team I mean a group of individuals with a common understanding working in unison toward a common goal.

Because of the bandwidth issues I described earlier, you cannot hold and maintain a common understanding and work in unison unless you are likely to pass by your teammates on the way to the restroom. That's why shared collaborative spaces are the best. That's also why teams that are split across floors experience many of the same problems as teams split across continents.

To run a successful distributed project, you must create clear boundaries between the distributed teams to provide enough isolation to allow them to work independently with only minimal coordination; the clearer the boundaries the better the result. I talk about this more in my column "Blessed isolation—Better design" in Chapter 6.

Eric Aside Establishing clear boundaries between the distributed teams doesn't prohibit interaction outside those boundaries. You can still do cross-team design and code reviews, planning, and brainstorming. Teams can help each other resolve issues and generally engage with each other. What clear boundaries do is keep teams from blocking and interfering daily. Instead, high-bandwidth communication is necessary perhaps only once a week, and cross-team activities become nice to have instead of must have all the time.

Typically, boundaries are architectural and isolate components, but they could be boundaries between versions or responsibilities. You could also have distributed teams focus on local market scenarios. Treat each distributed team like a dependency or a vendor and you'll be on the right track. This doesn't add unnecessary overhead; that communication overhead is there regardless. Instead, it reduces unnecessary conflicts, catastrophes, and cleanup.

But if your teams are isolated, how do you keep that sense of unity and sharing as you drive toward common goals? That's where the last big, bad B comes into play.

It would be so fine to see your face at my door

The third big, bad B is being there—being there to create the human bonds necessary to achieve real cooperation and connection.

Making the best use of limited bandwidth and clear boundaries will mitigate much of the difficulties with distributed development. However, you are still one group of people working on the same project with the same goals. There are important relationships you must maintain between teams, peers, and reporting structures that require a human face. Being there is important.

VTC and other live-presence tools can help. You should use them regularly for one-on-ones and meetings, perhaps not every time but at least once per month. Remember to reserve the overlap time between locations for these functions.

Of course, nothing can replace actual human contact, so plan to have everyone visit with each other at least twice a year. Plan one visit in October for the company meeting and performance reviews, and plan a second visit in February or March for budgeting and fiscal year planning. Don't visit for a few days; visit for a week or two. Have morale events, one-on-ones, and training sessions, as well as planning and review meetings. Make the most of your time together.

A number of teams do rotations or exchanges. In these cases, team members from one location spend three months, or even six months, with team members in another location. It can be a swap or an assignment. These rotations provide tremendous knowledge transfers while creating understanding, empathy, and connection across the groups.

One last thought.... Sometimes giving the illusion of being there can be very effective at maintaining relationships and improving communication. Some companies have tried continuous video feeds of common areas. A cheaper and fun solution that's been effective at Microsoft involves big pictures and even life-size cutouts of team members placed in high-traffic areas. As people leave meetings or walk the halls thinking about the project, they see one of these cutouts and are reminded to include that person or team in the conversation.

Where are you when the sun goes down?

With a little effort, you can make distributed development work for your projects and teams. Soon the sun may literally never set on your team. The advantages for you and Microsoft are enormous.

The diversity of experience, culture, and ideas might initially cause discomfort for you and your group, but over time it will make you better, your group better, and your products and services better. Your work will be more relevant in more markets for more people, and you will learn and grow in the process. Take the time to beat the Bs and you, your group, and your work will become worldly wonders.

December 1, 2008: "De-optimization"

Why? Why! Why do managers make stupid decisions that cause devastating churn and tawdry results? And it's not just managers, though they are particularly proficient at promoting poor performance—architects, leads, and individual contributors flood the lives of their teams with wasteful, useless, misdirected activities, leaving us even less opportunity to deliver real value. What reason is there for this farce? Simple. We are optimizing—optimizing our obsolescence.

What kind of idiot optimizes their own undoing? The ordinary kind. You do it, your friends do it, and your boss does it. It's all those good intentions that pave the way to disaster. We optimize the wrong behavior to achieve the wrong results. It’s wrong and avoidable, but hey, why think when you can cause mayhem with so little effort?

You want answers?

Let me save you some trouble and reading by giving you the answer first—**optimize for desired results**. It sounds simple and obvious, but people pervert that goal in so many imaginative ways that I better break it down word for word.

- **Optimize** Measure how good you are now, analyze how you could be better, alter your approach, and then measure again. Microsoft is great at optimizing, but we measure what’s handy rather than what matters. So, we optimize for the wrong result. You can read more about this in my column “How do you measure yourself?” in Chapter 2.
- **For** Have a purpose. Optimizing for the sake of optimizing is purely self-gratification—don’t do it in public. Instead, be deliberate about your purpose. Think it through. Know what you are doing. Be a professional. Wake up.
- **Desired** Focus on what you want, not what you don’t. This is a common trap. People optimize around the problem instead of the solution. Bureaucracies and slow software are built upon this misdirection. They focus on controlling people or code to prevent the wrong behavior. Their focus should be on making the right behavior fast and easy, and then catching the exceptions.
- **Results** Never optimize a step or algorithm in isolation. Instead, optimize the end result you seek. We have all experienced the impact of local versus global optimization. It kills our efficiency and innovation; it’s killing our planet. Yet over and over again, people can’t see beyond the problem at hand to consider the outcome they’re truly after.

Can you recognize when you are de-optimized? Let’s run through some examples and check.

Eric Aside I realize it’s a little confusing to talk about optimizing both code issues and people issues in the same column. I couldn’t resist because the number of similarities is startling. If it’s easier for you, just think about whichever problem you prefer.

I think I can handle this

How do you handle run-time errors in your code? How fast does your code run when no errors arise? Is it a smooth or bumpy ride for error-free operation? The fastest, simplest path through your code should be the 80% case, not the 20% case. However, that doesn’t mean you shortchange error handling; you just don’t optimize around it. Trust that your code will run error-free, making it run fast. Verify it was error-free, ensuring the right result. Trust but verify.

Likewise, how do you handle people and process errors? Do you check their every move? Do you have people do it your way, jump through hoops, and fill out redundant forms to ensure they aren't cheating? Or do you trust people to do the right thing—clearing the desired path of obstructions, and later verify that work was done properly? Trust but verify.

My altruistic readers, including managers, might claim that they do trust their coworkers. Really? How did you react the last time something went wrong? Did you quickly fix the root cause and move on, or did you start an inquisition randomizing your team for days or weeks? Do you micromanage or do you delegate? Do you specify every step or do you specify the result? Trust is hard. Luckily, you're being paid.

Déjà vu

How decoupled and cohesive is your code? Are the classes, functions, and functionality all intertwined and unmanageable, or are they independently testable and separable, each piece having its own purpose and task to perform?

Well-architected and layered code is far easier to test, maintain, and enhance. However, it doesn't perform quite as well as tightly coupled code. It's a tradeoff. If you optimize purely for speed, you eventually get unmaintainable spaghetti code. If you optimize purely for architecture, you can't be competitive in performance. How do you strike the right balance?

Most teams don't strike a balance between architecture and performance—they ride a rollercoaster:

1. The team starts with a nice architecture. It works great, and everyone feels good.
2. They optimize it for performance. Now it works better—the original team clearly wasn't as sophisticated.
3. The code is unmanageable, it can't be enhanced, and performance has hit boundaries, so the team painfully refactors the code. Now it's manageable again and everyone is happy—the prior team clearly were neophytes.
4. The performance isn't competitive, so the team optimizes again for performance. Now it's competitive again—the prior team clearly had lost its way.
5. Now the code is unmanageable, so return to step 3.

There's another variation—the code is so twisted that the team can't fathom refactoring. Their product cycle keeps getting longer, and the code keeps getting slower, requiring more memory and processing speed. That's a popular variation.

The right approach is to optimize for desired results—performant code that's easy to maintain and enhance. Instead of just measuring the speed (easy), you measure the speed and the code complexity. You seek the optimal balance of both. If you're really sophisticated, you'll also measure team health and customer satisfaction indicators, seeking a balance of all four. Wow, that's almost like running a business.

The beat of a different drummer

Let's try one more subtle example—product team structure. It's a war zone out there between traditional product development and the upstart Agile adherents. Who's right? Who cares! Never optimize around a step in isolation—optimize for desired results.

The desired result is delivering the most customer value in the shortest time. Remember, customer value is not measured by feature count; it's measured by delivering delightful end-to-end scenarios with high quality.

So how do you deliver high value quickly? You apply the Theory of Constraints (TOC). TOC says that the fastest way a project can accomplish anything is constrained by the slowest step. Say your user experience, content publishing, and operations teams are shared and can scale to your needs; your PM team can spec an average of four features in a month; your development team can code two features in a month; and your test team can validate three features in a month. There's not much point in your PM team going full speed, is there?

Yet managers will push the PM team to keep writing specs the dev team can't process—optimizing locally instead of globally. Adding people to speed up the dev team doesn't work either (note The Mythical Man-Month and the economy)—again, the focus is too narrow.

The right solution is to pace the PM and test teams to the dev team. Put in buffers to account for variability between features, but never have the PM and test teams outpace the dev team. This TOC strategy is called Drum-Buffer-Rope. Because it's hard to precisely predict the dev team's pace, you constrain the size of buffers, avoiding too much work in the dev team's queue should the situation change.

This is why feature crews work so well. You're optimizing for the desired result—working scenarios. In feature crews—an approach from Office—PM, dev, and test team members tie themselves to one piece of a scenario at a time till it's completely tested and integrated. They can't get ahead of each other. Versions of Scrum and eXtreme Programming work the same way. It's not the combined teams that are essential (though communication is easier); it's the pacing of work together that optimizes the delivery of complete, high-quality customer value.

Eric Aside A reader pointed out that often the key constraint in software development is communication bandwidth. The best way to increase communication bandwidth is to co-locate teams and have them work together end-to-end on features. This same approach also works to improve pacing, as I mentioned above. Some of my Scrum teams are also switching to a Kanban model, which directly applies Drum-Buffer-Rope in a simple and intuitive manner.

Don't panic

It's so easy to get caught up in the immediate and optimize around the issues directly in front of your face, instead of the ones you actually care about. People do it all the time—I guess we're programmed instinctively that way. That's a perfectly good reason to optimize the wrong behavior for the wrong results, but it's a poor excuse.

You should know better, and if you don't, you have no right to draw a paycheck. Consider the result you desire to achieve, think it through, measure a balance of factors, and optimize as a whole. It's not that difficult. We attempt it every day as we balance our lives. The key is to be deliberate rather than to juggle; to plan rather than panic. You can do it if you simply keep your sights on the finish line.

April 1, 2009: "Your World. Easier"

During difficult economic times like these, people tend to whine less about common complaints that now seem trite. Mostly, I'm relieved not to hear how much e-mail is in Ingrid's Inbox, how Brian broke the build again, and how Suresh's service schedule slipped successive sprints.

However, it's during difficult days that we should patch plaguing problems. When are you going to be more motivated to mend malignant maladies? Surely, no additional alliteration is advisable.

A surprising number of common issues can be solved using two simple techniques—single-piece flow and checklists. There's a ton of behavioral and process theory behind why these simple methods are effective. The point is that they are effective, and you and your team are less effective without them.

All too easy

Take Ingrid's Inbox. Like most Inboxes, Ingrid's is overflowing. She spends tons of time on it, yet it only gets bigger. She constantly loses track of mail, discovers mail, and revisits mail. It's hopeless.

Ingrid would have her mail under control if she followed single-piece flow. Single-piece flow tells her to handle one piece (one message) at a time till it's done. By "done" I mean she'll never look at that message again (except to answer a related message).

Here's how it works. Each time Ingrid reads an e-mail message she takes one of four actions:

1. She deletes the message (my favorite).
2. She files it away in a folder.

3. She forwards the message to someone else and then deletes or files it.

> **Eric Aside** How do you ensure follow-up to an important forwarded message? I use one of two approaches.
>
> - ❑ I put my forwarded mail into a special folder for more intensive attention. The message stays there until I get a response or send a reminder (the reminder then moves to the special folder). I use this technique when there isn't a specific or critical deadline.
> - ❑ I move my forwarded mail into my calendar, where it becomes an appointment to remind the recipients to reply or to close out the issue. I use this technique when there is a specific or critical deadline.

4. She answers the message and then deletes or files it.

That's it. She never opens a message and then leaves it in her Inbox. By the end of each day, every day, her Inbox is empty. She never misses a mail message, loses a message, or revisits a message.

Single-piece flow is efficient because it removes overhead and rework. There's overhead every time you context-switch to look at a new message, and there's rework when you reread a message. In single-piece flow, overhead is minimized and rework is eradicated.

> **Eric Aside** Sure, there are exceptions to the read-it-once rule, but they account for less than 5% of the mail I get in a day. Even those e-mail messages get filed in a special folder for more intensive attention. I also use Inbox rules to prefilter mail from discussion groups.
>
> If you're wondering how to get started and don't want to just delete all the mail in your Inbox, follow these steps:
>
> 1. Take mail you are actively working on right now and move it into a special folder.
> 2. Create folders for your remaining mail that correspond to your obligations and interests.
> 3. Search your Inbox for mail that fits each folder and move that mail into the folder.
> 4. Go through your special folder and clear out 90% of it using single-piece flow.
>
> Now you are on your way.

Déjà vu—all over again

Brian keeps breaking the build. You can punish Brian till he's afraid of checking in code regularly, but doing so only causes other problems.

A better solution is to give Brian a checklist. Checklists are wildly misunderstood, improperly developed, and underutilized. Regrettably, well-meaning, compulsive people list everything possible Brian should check before he submits code to the build. That's not only a waste of time, it's also ineffective.

Brian's checklist should list only common causes of build breaks (less than one page's worth). It should be in Brian's sight when he submits code. The goal is to be quick, easy, and effective.

Too long or complex, and Brian won't follow it. Too short, and it's not effective. Luckily, most mistakes are common mistakes. Thus, all the team needs to do is collect a list of the common or critical causes for build breaks and turn that into a checklist. The same is true for design review lists, code review lists, and all checklists.

Remember to update your checklists as your failure patterns change, or they will become stale. Checklists prevent common errors and promote good habits. Any structured, manual process you follow should have a simple checklist—unless you like being Brian the build breaker.

Eric Aside You may have battles deciding what goes in a checklist. You shouldn't. Remember, you list what the data says are the common or critical failure points, not everything that ever happened.

For example, I added a checklist to my e-mail signature several months ago, which I check and then delete before I send every mail. It lists the two mistakes I've made for years, but haven't made since:

- ❑ Check the Cc and Bcc lines for undesired aliases
- ❑ Ensure the subject line is accurate

Slip sliding away

Suresh's software squad slipped their schedule on successive sprints. Bad news for a service—or any project. Is it time to work weekends? No. Is it time to slap the squad silly? No. Is it time for single-piece flow and checklists? Yes.

Suresh's squad is made up the usual way—a few PMs, including a service engineer; a bunch of developers; and a similar-size bunch of testers. The PMs write specs, the developers code them, and the testers test them. The squad is using Scrum-like sprints with a nicely prioritized backlog of features.

Unfortunately, the PMs are creating specs faster than the specs can be implemented. They waste time and effort on specs that change or are cut before they see daylight. The developers and testers jump from feature to feature as they get blocked. Nothing is in sync. Nothing gets finished. The schedule slips incessantly. In retrospectives, all Suresh's squad talks about is unblocking people.

Even though Suresh's squad is using Scrum-like sprints, they aren't using single-piece flow. They aren't splitting into cross-discipline teams, with each team working on one feature at a time till it's done. They don't even have a checklist that defines done. It's doomed.

Once they create feature teams that spec, code, and test one feature at a time (sometimes called *feature crews*) and a checklist that defines done, there's a chance for progress. The single-piece flow removes blocking issues because everyone is working on the same problem. Instead of jumping ahead, the checklist keeps the team honest, motivating them to work together, finish, and stay focused. Now the team doesn't waste time context switching and can tell how long it really takes to complete a feature, leading to confident scheduling and higher quality finished products and services.

Eric Aside Many people wonder what to do with feature crew PMs once they finish specing, or developers once they hit code complete. For PMs, there are two solutions—be part of multiple feature crews or be prepared to work with dev and test peers on problem solving and development. For developers, the right solution is to work with the test team on tools, automation, unit tests, and component tests.

There's another clever solution written up by a former Microsoft employee, Corey Ladas. He calls it *Scrumban,* and you can read about it on his and Bernie Thompson's Lean Software Engineering site (*leansoftwareengineering.com/ksse/scrum-ban/*).

Our two weapons are

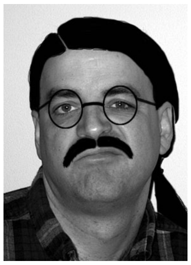
So there you have it—single-piece flow and checklists. Two enormously useful, remarkably simple, and yet woefully underutilized techniques for managing workload and building quality software.

Single-piece flow and checklists can be applied to individuals, small teams, and large divisions. They aren't controversial when used pragmatically, and have years of documented case history supporting their effectiveness.

Sure, you could create a whole grassroots movement around single-piece flow and checklists, but that seems a bit overblown for such simple ideas. Maybe you should just use them wisely. Enjoy the time you get back and the improvement in your results. The best things in life are often the most basic and simple.

Eric Aside James Waletzky has an amusing blog entry with a few checklist references, entitled "Checklist? We don't need no stinking checklist!" (*http://go.microsoft.com/FWLink/?Linkid=219829*).

April 1, 2011: "You have to make a decision"



What's worse—a flawed decision or no decision? That's easy. Decisions keep a business moving. An imperfect decision might move your business slightly in the wrong direction, but at least it will be moving. Make a few adjustments, and you're back on track.

Making no decision brings business to a halt. Even if you make a great decision during the delay, the time needed to regain momentum will leave you far behind. Like boulders and trains, groups of people have inertia. It takes great effort to get them going, so it's better to keep them moving forward and adjust, rather than halt progress and restart.

Unfortunately, leaders often don't have enough information to make a clear decision. What do you call leaders who are indecisive when dealing with ambiguity? Incompetent, ineffective imposters. They are losers, not leaders. If you want to lead, you better know how to make good decisions with partial data.

I'm the decider

Before discussing making decisions with incomplete data, I should point out that being a decisive dunce is no better than being an ineffective imposter. Decisive dunces make decisions quickly (good), but without clear thinking or cause (bad). This results in random or backward progress and an exasperated team. Having too little information is no excuse for being arbitrary or clueless.

Instead, you want to inform yourself as much as time will allow and then make your decision based on context and experience. How much time do you have? Only until the point your team becomes blocked. How do you apply context and experience? Develop a consistent framework for making your decisions that helps you spot patterns, and then rely on your well-informed intuition to make the best choice.

This is easier to describe with examples. Let's talk about hiring decisions, product tradeoffs, and triage decisions.

Eric Aside If you want more time to make decisions, then plan ahead. Map your business rhythm to a calendar, and begin your decision making early in anticipation of events.

If you want better data to make decisions, then put measurements in place, utilize business intelligence, and be better informed.

Both strategies for better decisions require forethought. Don't like a vacuum? Fill it.

To hire or not to hire

I've been an "as appropriate" for a decade, meaning that I am the last interviewer on an interview loop. My role is to make the final hiring decision about the candidate. Sometimes that decision is very easy because the prior interview comments are compelling in one direction or another and they fit my personal assessment of the candidate well.

However, the hiring decision is often difficult. The prior interview feedback is mixed and inconclusive. For an interview, I have at most one hour to gather information directly from the candidate. In addition, the recruiting team and hiring manager are blocked awaiting the results of my interview, so I have only an hour or two to decide.

To make the best possible decision, I follow the same routine each interview. I prepare before the interview by reading the candidate's resume, checking out any online content it references, and reading the prior interview feedback. I look for patterns in the comments that might indicate the root cause(s) of concerns. Then I interview the candidate with the goal of confirming the root cause(s) and determining whether or not the concerns can be overcome.

Now I've collected all the facts I possibly can without significantly delaying the offer. I've done the analysis, and my intuition is informed. How do I put this all together? I step back and let my intuition guide me. After all the analysis, and all my years of experience, what decision seems right?

As Malcolm Gladwell points out in Blink (*www.gladwell.com/blink/index.html*), an informed intuition is an exceptional instrument. I'm not talking about judging a candidate from a quick glance. I'm talking about learning all I can and then, instead of focusing only on the individual details, stepping back and looking at the candidate as a whole with everything I know about her and the other candidates I've met. That's hard to do consciously—your informed intuition can be your guide.

Eric Aside Even if I did take time to call every reference, bring the candidate back for more interviews, and do a background search online, my decision might not be much better. As Gladwell also points out, sometimes the extra analysis and second guessing is worse because you start to ignore your overall assessment and cumulative experience in favor of individual facts.

If I'm still undecided about a candidate, the decision is easy—don't hire the candidate. You always want to feel confident about the candidate's potential for success for the sake of the candidate, the team, and the company.

Now consider the alternative

Product tradeoffs can be very difficult to make. There are often many opposing viewpoints, but there's rarely conclusive data. Great data from telemetry and web services can make a huge difference, but data informs—it doesn't decide (necessary, but not sufficient). You must make the call quickly based on the data and your collective experience or risk losing your market position.

The individual decision may be about a user story, an architectural approach, or even a development tool. Regardless, there will be pros and cons for each choice. First, use your value proposition, architecture principles, or other relevant framework to clarify your options. If you still have a number of valid choices, you can quickly analyze your tradeoffs by using one of my favorite tools, Pugh Concept Selection—a fancy name for a simple technique. (I described it briefly in "My experiment worked! (Prototyping)" in Chapter 6.)

In Pugh Concept Selection, you make a table in which the rows are choices and the columns are decision criteria. (Excel is perfect for doing this.) The result is like a pros and cons table with a twist. Instead of indicating which choices meet the criteria best on some absolute scale, you pick one choice as your default and give it zeros for all criteria. Then you rate all the other choices *relative* to your default choice. If a choice is better than the default for a criterion, you put a +1 in that row and column. If a choice is worse, you fill in -1, and if it's about the same, you fill in 0. Add up all the columns, and you have the rating for each row. Now you can see how all the choices compare.

Eric Aside You can weight criteria (columns) differently to indicate what's most important. Typically, people use power series for weights (1, 3, 9). To determine the sum of a row, you multiply its ratings (-1, 0, +1) by their associated weights and add them together. The Excel SUMPRODUCT function does this nicely. You'll find a sample spreadsheet (Pugh Concept Selection Example.xlsx) in the online materials.

Filling out the table of choices as a group can be fun and insightful. You learn a great deal about what the tradeoffs between choices really are and what you care about most. However, Pugh Concept Selection won't make your choice for you (though it could reject a bunch). While one option might get the best score, there are likely several top options worth considering.

Once the table of choices is complete and you've talked through the results and insights gained, it's time to make a decision. That's when you throw out the table and rely on your newly informed intuition. The analysis is necessary and instructive, but it's your experience and overall context—your intuition—that can take that analysis and sense the right decision.

Eric Aside "But won't your decision contain bias?" You bet it will, just like every decision you ever make in your life. Being well informed and considering all reasonable alternatives helps you remove as much bias as possible. However, your final decision will always reflect your unique perspective.

Tell me why

When you are managing product issues toward the end of a cycle, there are many issues to cover, so decisions must be made quickly. In addition, you are usually making those decisions in a group of at least three people representing different interests, so you must reach a consensus in which no one objects.

Eric Aside At Microsoft, this decision-making process is called triage. It can be one of the most divisive activities in the development cycle if done improperly.

How can you quickly convince three or more people to agree with confidence about important issues? You need three things:

- **The root of the issue** Without knowing the root cause, you can't apply your past experience to the problem with confidence.
- **The scope of the issue** Without knowing how many customers and partners are impacted and how they are impacted, you can't judge the benefit of acting.
- **The risk of the issue** Without knowing the risks involved, you can't balance the costs against the benefits.

Once you've understood these three things, which every late-stage issue report should contain, you've informed your collective intuition. Now the group decision makers can depend upon their visceral reaction to a suggested resolution. Of course, that's no guarantee you'll agree. For more on my framework for these kinds of tense group decisions, read my column "Are we having fun yet? The joy of triage" in Chapter 1.

Wait, there's more

"But what if more information arrives later that significantly changes your thinking?" No problem. We want to make a decision that lets us move forward, not necessarily a perfect decision. The way we gain insight and experience is to try, receive feedback, adjust, and try again.

That's no reason to change decisions constantly, however. To maintain your forward momentum, you should iterate, making slight adjustments as you go. Yet occasionally there are enough new internal insights or external expectations to suggest a substantial course correction. That's okay—it's what makes our lives interesting and our business engaging.

Eric Aside You can read more about what happens when managers constantly change their minds in "Spontaneous combustion of rancid management" in Chapter 9.

Each time you analyze new information, your intuition becomes smarter. You get more experience and a better perspective. You make better decisions. Keep an open mind, listen well to customers and contrarians, apply a consistent framework, and trust your well-informed instincts. Don't wait to be certain. Decide, iterate, and constantly lead your team closer to its goals.

Chapter 4

Cross Disciplines

In this chapter:

Software development, when done well, requires a broad skill set. You need to truly understand the customer and the business. You need strong user experience design skills and knowledge of usability (even for API work). You need engineering design skills, software development skills, software testing skills, and tremendous familiarity with the target platform, which could be a server farm.

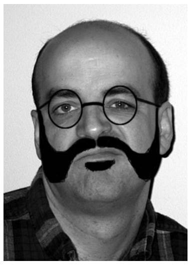
Sure, you might have all those skills, but how good are you at talking to yourself? If you lack that skill, you'll likely miss important aspects of product development. If you're great at talking to yourself, that could lead to even more serious problems. In all, you are better off developing an appreciation and knack for working well with others.

In this chapter, I. M. Wright tackles relationships between developers and other disciplines. The first column describes the symbiotic relationship with test. The second delves deeper into the role of testers. The third column deals directly with the Achilles Heel of most engineers: how to interact with nontechnical people. The fourth questions why and when different disciplines are necessary. The fifth covers the contentious concerns of sustained engineering. The last column pleads the case for a senior test staff commensurate with development and program management.

I spent the first half of my life wondering why people couldn't be more like computers. I've spent the second half being thankful they aren't. Sure, I still get frustrated at times with people who don't think the way I do, but my life is more interesting and my solutions are more creative thanks to the diversity of people that surround me. It's part of why I love the "Hard Code" column; the response is never predictable or boring.

—Eric

April 1, 2002: "The modern odd couple? Dev and Test"



Is there a more classic love/hate relationship than the one between developers and testers? (Okay, maybe between developers and program managers, but I digress.) As a developer, you either see testers as nagging or persistent, nitpicky or thorough, obnoxious or passionate, unsophisticated or customer-focused, doomsayers or cautious.

The same could be said for how testers view developers: brilliant or geniuses, dedicated or hard-working, creative or inventive—uh yeah, right. If the truth be told, testers often think as poorly of developers as we do of them.

Many teams are far from reaching parity and mutual respect between these disciplines. Some steroid-sucking developers out there might respond, "That's fine—lead, follow, or get out of the way. If the test team can't keep pace, can't stand toe to toe, can't flex their own muscle, then they should step aside and avoid the oncoming dev machine."

Big words from small minds. It's time to tear this twisted testosterone tale apart, piece by piece, and reveal how real developers should be working with the greatest allies they have: testers.

How do I love thee? Let me count the ways

So why do development teams often treat their test teams as second-rate and unworthy of respect and cooperation? I'll break down the reasons, one at a time:

- **All testers really wish they were developers; therefore, developers must be better.** Any ridiculous generalization like this can't be true. Not all testers wish they were developers. Some do, certainly, but others want to be golf pros, racecar drivers, product unit managers, program managers, parents, priests—you name it. Some even want to be great testers.

  The real disgrace about this myth is that it shows how insecure some developers really are. A developer who is truly proud of his work and accomplishments wouldn't need to feel that development is a superior discipline.

- **Testers are not as smart as developers.** Taken as a corollary to the previous fallacy, this is often voiced as, "If testers were as smart as developers, they'd be developers." Again, bull excrement at the most basic level.

  However, even fair-minded people will point out that testers often aren't required to have the same level of education as developers. Black-box testers, in particular, are frequently hired with little or no technical experience. To me, this is purely a case of the chicken versus the egg—there simply isn't enough external training for testers.

Eric Aside Black-box testing treats the product as a black box. You don't have any access or knowledge of the internal workings, so you probe it like a customer would—you use it and abuse it till it breaks. Microsoft has been steadily moving toward white-box testing, in which you use exposed instrumentation to automatically and systematically test every aspect of the product.

Many people inside the company are working hard to correct this inequity through initiatives like the Readiness at Microsoft Program (RAMP) and the Test Lead program. Even a few universities and local community colleges have begun offering courses in testing. But until colleges and universities create standardized bachelor's and master's degree programs in this important discipline, testers won't find quality opportunities to learn more about their field through higher education.

Does this lack of educational opportunity mean that most testers are not smart? Of course not. There are strong and weak testers just as there are strong and weak developers. That's all.

- **Testers wouldn't have much to do without developers.** The idea that without developers, testers would have nothing to test leads some people to think that developers are better. (Of course, the fact that developers often generate plenty of work in the form of bugs leads many people to think that developers are worse.) Nonetheless, is every upstream discipline considered inherently stronger than the downstream counterparts who depend on them? It is harder to direct a movie than to run the projector, but then again, it's easier to dig up a diamond than to cut and polish it.

  It really comes down to the complexity of the work, and testing products and functionality well is easily as difficult as writing the stuff in the first place. Many of the hard theoretical coding problems have been solved, but many of the hard theoretical testing problems are still waiting for answers.

- **There aren't as many strong test teams as there are strong development teams at Microsoft.** Based on this presumption, developers might extrapolate that their discipline must be better than testers in general. This is a bad assumption and poor pretext for not doing all that you can do to grow your test team and strengthen your whole group in the process. Testing is not as mature a discipline as development. Using this as an excuse to act condescendingly toward testers only serves to lower a developer's maturity level rather than raise a tester's.

Necessary evil or priceless partner?

We can't ship a product without testing it. Think of the three worst bugs that testers found in your code last cycle and you'll know what I'm saying. What many developers haven't realized yet is how well you can ship a product with the test team's support.

Unfortunately, for many groups, testing remains a necessary evil rather than an integral priceless partner. I know that there are developers out there chiding me saying, “Integral priceless partner, as if! Test is what they’ve always been: monkeys that bang on the code till it breaks and then whine about it till we fix it.”

If that’s the way you treat them, then that’s what your testers will be. However, there’s an alternative approach.

A man’s got to know his limitations

It starts with understanding three principal areas where development falls short: writing perfect code, understanding RAID holistically, and working within the customer’s environment.

No developer writes bug-free code; even if there are no logic bugs, there may be behavior bugs. No developer lives and breathes RAID; he just uses it as a tool. No developer commonly works in the customer’s environment; he works on big machines with tons of memory, processing power, and disk space. Developers also have high privileges, fast networks, the latest operating system and patches, and no legacy code—and they work in their native language. Basically, developers are hopelessly out of touch with what is necessary to move the product from their desk to the customer’s.

So guess who’s holding the safety net and providing the balancing pole for developers as they cross the tightrope between code complete and release? That’s right, it’s your testers.

Testers find both the logic and behavior bugs; they live and breathe RAID; they know where you stand and how much work needs to be done; their computers run all the time in the customer’s environment and in all the different languages, platforms, and configurations. Testers can tell you how many bugs you need to fix each week without guessing. They know where the problem areas are and can give you gut and metric readings of how far you need to go.

You complete me

Developers can make you say, “Wow!” But testers will save your behind and make you proud of your development efforts.

If you treat your test team like trash, trash is likely what you’ll ship and trashy is likely how you’ll feel. If you prefer a smooth release and want to ship a great product, make your test team your ally.

The key is for you and your development team to understand and appreciate all that the test team offers you. They cover the areas that you can’t or don’t want to do. They keep you on schedule and on track. They keep you honest and represent the customer’s perspective.

Tell testers how valuable they are to you. Tell them how they can help both your teams ship a great product. Do everything that you can to support them, and they will come through for you.

Eric Aside I really can't say enough about my respect and appreciation for the test discipline. In the nine years since I wrote this column, some nice strides have been taken at creating parity between dev and test in all areas, but there's still far more we can do, as I discuss in a later column in this chapter, "Test don't get no respect." As for managers and PMs, I admire and respect them too, but since I live under the darkening shadow of their every whim, forgive me for not stroking their egos.

By the way, if you're already making the test team a valued partner, it's time to do the same with service operations. The same arguments apply, so why not trade in your pain and frustration for synergy and success?

July 1, 2004: "Feeling testy—The role of testers"

I've been carping on code construction quality quite a bit lately. One of the five key methods of removing bugs early is unit testing. (The others are design, design review, code review, and code analysis, like PREfast.) Doing comprehensive unit testing has drawn dubious disbelief from some devs I meet: "Isn't testing a job for testers?" "If devs write the [unit] tests, what's left for testers to do?"

Eric Aside PREfast is a static analysis tool for the C and C++ programming languages that identifies suspect coding patterns that might lead to buffer overruns or other serious programming errors. Though initially used only internally, it shipped as part of Microsoft Visual Studio 2005.

First of all, unit tests are usually focused on isolated components, and testers cover far more than component testing. Their tests include boundary coverage, structural coverage, path coverage, black- and white-hat testing, and a host of system-wide and scenario-based testing, just to scratch the surface.

No, first of all, what the heck do you care if there's nothing left for testers to do anyway? Since when have you abandoned your responsibility to check your own work? Is passing on crappy code some kind of perverse form of charity to you? Do you not give a flying fork about doing your job? Have you no decency or pride?

Eric Aside When I wrote this column, I initially got stuck for a "second of all" paragraph, so I reread the first "First of all" paragraph. Suddenly, the "No, first of all" paragraph flowed out faster than I could type it. That's always a good thing.

Unit testing drives better implementation design; more testable code; fewer regressions, build breaks, and BVT failures to debug; and better overall construction quality.

Advanced protection

In my article called "Where's the beef? Why we need quality" (see Chapter 5), here's what I said about a dev's responsibility to write instrumentation and unit tests:

> *NO, this is not a tester's job. Test's job is to protect the customer from whatever you miss—despite your best efforts. The testing team is not a crutch on which you get to balance your two tons of lousy code. You should be embarrassed if test finds a bug. Every bug that they find is a bug that you missed, and it better not be because you were lazy, apathetic, or incompetent.*

However, in fairness to the devs that actually care enough about their peers to consider the impact of unit testing on test jobs, I should discuss more about the role of testers in protecting our customers and how testing may actually evolve if and when devs finally get their acts together.

A change will do you good

In "The modern odd couple? Dev and Test" (which appeared earlier in this chapter), I pointed out three primary ways testers protect our customers:

- Finding bugs we miss.
- Living and breathing quality metrics (mostly from Product Studio).
- Running tests in the customer's environment as opposed to the developer's environment.

Devs writing unit tests doesn't change any of these roles for testers.

But testing is changing nonetheless. Almost all new test hires are expected to know how to write automation code and white-box tests. The more cynical among us may think that there is a secret plan to eliminate testers after all tests are automated. However, the real reason for enhanced automation resides in the time it takes to run all the necessary tests to pass an urgent security, privacy, or reliability update, or even to just validate a build. As teams try to push quality upstream, automated tests are essential to ensure construction quality before check-in.

That brings us to the next key turning point. As the quality of your code at check-in increases, how testers perform their role changes. Note that their role is the same—find missed bugs, analyze quality metrics, and run in the customer's environment. It's how they do it that's different.

The twilight zone

Right now finding bugs in checked-in code is akin to spotting a coffee shop in Seattle—they are hard to miss. Sure, you write a test plan and test cases to exhaust all the different possibilities. But the expectation is always, "Just run the application, the bugs are sure to be there."

As teams push quality upstream, finding bugs after check-in becomes more difficult. At first, teams simply test *buddy drops* as part of their check-in criteria. A number of teams are doing this now, including some in established groups such as Windows and Office.

Eric Aside A buddy drop is a private build of a product used to verify code changes before they have been checked into the main code base. That way the impact of unstable code on other teams is minimized. Developers share the private build only with their "buddies"—that is, their teammates.

But as dev teams begin to use disciplined practices and measurements to predict and control their bug counts, the number of bugs found in a buddy drop should plummet by a factor of a thousand. Thus, the typical Microsoft team of 15–20 devs that produces 3,000–5,000 test bugs per year would instead produce 3–5 test bugs per year. When buddy builds get that solid, test needs to learn some new tricks. (I talk about how some teams have achieved these low bug counts using cool software engineering principles in "A software odyssey—From craft to engineering," which appears in Chapter 5.)

Commander Data

Remember, test's role is to protect customers by finding bugs that devs miss, analyzing quality metrics, and running in the customer's environment. In a world where code handed to test has only a handful of total bugs, the challenge is to take test's three responsibilities to the next level.

With such low bug rates, testers should no longer be able to easily find bugs in isolated components or common configurations. Instead, they need to focus on full customer system scenarios and realistic customer configurations, further leveraging their unique customer perspective. While test teams do this a fair amount today, it likely will become their primary focus in the future.

The other key change is in analyzing quality metrics. For devs to get low bug rates, they must collect quality data from the beginning of design, through compile and build, and then use it to know what mistakes they are making, how best to find them, and when they can confidently say they've found 99.9% of them. Someone outside of dev needs to be checking the dev's assumptions, ensuring that the data is reliable and accurate, keeping devs honest and customers protected. Of course, that's the ideal job for test.

A side effect of this rigorous dev process is that test will have far more quality data, in far greater detail, than they've ever had from Product Studio. The more data you have, the more there is to analyze and discover. You can almost hear testers salivating.

Eric Aside Most good testers I know are data fanatics. A perfect afternoon for them is spent up to their eyeballs in Excel spreadsheets analyzing data in different ways.

While devs will do some analysis, this is really a showcase area for testers that entails

- Conducting in-depth statistical studies.
- Noticing key weak spots and trends.
- Finding new ways to improve quality or efficacy even further.

Testers can become the quality process and analysis kings in this new world.

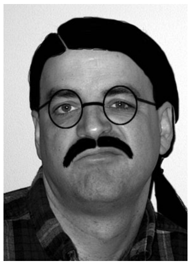
It's quite cool—I assure you

When devs graduate to engineers, testers can start leveraging comprehensive data and process analysis to assure quality. In others words, test becomes quality assurance. Quality control testing doesn't go away, but quality assurance grows into a major role. This same pattern has happened in other industries and is due to happen to software. If we don't make it happen, we will get beat.

So sure, most of this seems like a pipe dream now. Even though some teams are starting to see this level of quality engineering, it's a long way from being part of our largest systems (like Windows). But if your team is beginning to make headway, or if you know forward-looking testers who want to lead the next wave of software quality, suggest that they start learning about in-depth statistical data and process analysis. It could prepare them to take our game to a whole new level.

Eric Aside While we haven't yet gone from five thousand bugs down to five for our large complex projects, the focus on quality assurance and data is now happening in a number of divisions. One particularly interesting case is Bing. Bing changes daily (and I don't just mean the background picture). If test were trying to do pure quality control, the test matrix would be astounding. Instead, Bing test does quality control in key areas and spot checks for quality assurance in the rest. That includes intense aggregate data analysis, where a substantial portion of Bing engineering is directed.

May 1, 2005: “Fuzzy logic—The liberal arts”



All my life, I’ve lived among the willfully ignorant—people who might consider knocking wood a silly superstition, yet have no idea how their TV works, how planes fly, or how phones connect. To them, it’s all magic. They make up their own mythologies and rituals for getting technology to function. Then these neophytes have the gall to tell you to turn off the lights before you reboot or it won’t work.

Before I graduated from college, I had the perfect solution for dealing with these naïve fools. I simply avoided and ignored them. Hanging around with techies was right in my comfort zone. That is until I married a “fuzzy”—a liberal arts major. All of a sudden, learning to communicate with the technologically superstitious became enlightened self-interest. We’ve been together for 20 years now, and I’m beginning to get the hang of it.

Eric Aside Oh, the grief I got for this column (along with tons of praise). The grief was from techies who were private fuzzies and fuzzies who love technology. Both hated the stereotypes I drew. (Also, people who weren’t fuzzy or techie, like art designers, felt left out of the discussion.) As I say below, “Naturally, I’ve over-generalized here.” People are often too polite or hesitant to discuss contentious issues loaded with ambiguity. I over-generalize in every column to bring out salient points and drive dialogue.

It takes all kinds

Why should interacting with the ignorant be any concern of yours? Why not leave the fools to the foolish? I’ve got three reasons: coworkers, managers, and customers.

Sure, we hire folks for their technical savvy, but we all know that there are people who get it and people who tread water. For every one PM or manager who gets it, there are 5 to 10 who don’t. There are a lot of PMs and managers at Microsoft. Any improvements you want to make to your products or practices need to go through them. If you can’t communicate with PMs and managers effectively, I hope your geek past has sufficiently prepared you for a life of frustration.

Eric Aside Many PMs and managers come from the techie ranks. At one point, they understood all the details and took nothing for granted. However, over time it becomes expedient to let go of the details and think more about nontechnical aspects and the “big picture.” The world becomes fuzzy. Yes, I’m guilty of this myself.

As for customers, well, we don’t get to choose. Customers are customers, and the inability to effectively speak to a customer is career limiting. As I mentioned in my column “Customer dissatisfaction," which appears in Chapter 2, talking to customers is the key to making the right choices for critical product decisions. Customers don't like to be patronized or made to feel small or stupid. I suppose managers and PMs are the same way.

You've got to understand fuzzy folks. You've got to appeal to their best judgment in a way that makes sense to them, makes them feel smart and in control. The alternatives aren't pleasant.

They're not like us

Liberal arts majors are not like us. It isn't just the schools they attended or the classes they took. It's a whole different way of looking at the world that you need to truly grok (a word that might not be familiar to them).

As luck would have it, I've spent the last 20 years trying to understand fuzzy logic. I've discovered some key differences you need to internalize:

- **Liberal arts majors believe rules are rules.** Techies believe rules aren't meant to be followed blindly; they are meant to be analyzed and understood, then used or altered as needed. Fuzzies believe rules are meant to protect you, and they simply must be obeyed. Even worse, a fuzzy's version of the rules may not match yours. Just remember, if you plan to question or break any perceived rules in front of a fuzzy, you'd better be prepared to explain why it's safe and have an authority figure to back you up. Why? Because…
- **Liberal arts majors respect authority.** Techies typically don't respect authority, though they do respect achievement. So it might not occur to you that manager approval is that big a deal—but it is to a fuzzy. The good news is that most fuzzies consider techies authorities on technology, so they'll believe most anything you say in your area of expertise. Try not to abuse that advantage. Fuzzies may be different, but that doesn't mean they are stupid or foolish.

> **Eric Aside** At this point, you might be saying, "Wait a minute, fuzzies have been breaking rules and disrespecting authority for years." They have when they feel justified. In everyday life, fuzzies tend to prefer obedience to uncertainty. Techies prefer to rely on their own reason and logic to determine the rules and authority.

- **Liberal arts majors don't tinker.** Techies love to tinker; fuzzies avoid tinkering. Tinkering is breaking the rules. It feels risky and unsafe. This difference in attitude is subtle but very important. Fuzzies won't just try stuff. They won't right-click, press and hold, or try different menu items just to see what happens. So don't expect fuzzies to experiment unless they truly know it is safe. Likewise, don't expect a fuzzy to approve a change unless it's well worth the risk and has a safe abort.

- **Liberal arts majors assume everything is simple.** Techies know nothing is simple because techies focus on the details. Fuzzies focus on the larger picture where everything is simple, and if it isn't, it should be. Neither view is wrong. Everything you do should be conceptually simple and easy to explain at a high level, or chances are good that it will collapse under its own weight. Yet, the devil is in the details. Simple is hard for technical folks who thrive in the details. But...
- **Liberal arts majors don't care about the minutiae.** Techies love the minutiae; often it's the best part of a project. However, because fuzzies focus on the larger picture, tiny details only confuse the issues at hand. Therefore, if you are describing your idea or project to a fuzzy, you must leave out the minutiae and articulate the simple, high-level concepts and requirements behind your work. Otherwise, you'd better be prepared for no support, and rightfully so. Fuzzies aren't simpletons who need to be babied. They are integrators who will tie your work into everything else going on, if you can provide a clear and simple picture of how your work fits in.
- **Liberal arts majors are not concerned with purity.** Techies love purity. To them, it is beauty and grace. Purity washes away all the ugly tidbits, leaving the simple core truth of the problem. Unfortunately, all this escapes your typical fuzzy. Fuzzies don't care about the ugly details in the first place. They expect things to be pure and simple. Telling a fuzzy that you've found an elegant, simple solution is likely to get a response like, "Yeah, I should hope so, what did you have before?" If you want to convince a fuzzy to adopt your elegant architecture, don't argue the purity. Instead, talk to them about the customer or business benefits it provides like more reliability and easier maintenance.
- **Liberal arts majors care about feelings and appearances.** Techies typically don't even realize that feelings and appearances exist. It took my wife years of pointing out the importance of these things before I started understanding them. Fuzzies care tremendously about feelings and appearances. I know that seems stupid and counterproductive, but there just isn't any way around it, so don't bother arguing. Instead, when you propose an idea or plan to a fuzzy, be sure to consider how people will feel about it (assume that everyone is a fuzzy, which is how fuzzies think). Will anyone need to save face? Are you crossing into other people's territory? Are you contradicting someone in authority? You don't need to solve all these issues, but you do need to think about them and point them out to fuzzies. They will be impressed by how perceptive you are and then help you solve the people issues.

Naturally, I've over-generalized here. Not all liberal arts majors have these traits any more than all techies have the opposite traits. But you can't assume everyone thinks the way you do. Just putting your preconceived notions on hold can go a long way toward achieving clear communication.

Getting past security

One of the more important implications of these qualities that make liberal arts majors different is that they tend to surround and protect people in authority. Because fuzzies respect authority and care about feelings and appearances, including their own, they can't let just anyone talk to a senior manager or key customer contact. You've got to work through them. Sneaking past the fuzzy security may be fun and effective the first time, but when they realize what happened, the literary lynch squad will be offended and won't forget it.

Luckily, there are ways to soft-talk your way through. Explain why it's important to talk to the customer or manager. Allow the fuzzy to introduce you and set the stage (appearances, rules). Collect your issues and bring them all at once so the customer's or manager's time is respected and appreciated (feelings, respect for authority). Unless you are asked directly, leave out all the gritty details and cool design (minutiae, purity). Give the customer or manager clear choices of action (simplicity).

Management will love you; the customer will love you; and the crack fuzzy security force will love you.

Making things happen

If you want to drive engineering improvements on your team, you'll need to convince the fuzzies. This isn't easy because improvement means change, change means breaking the rules, and rules are rules. However, you can be an effective driver for change by following this simple strategy:

- First, describe the problem you're trying to solve. Use statistics to make it sound horrible. (It probably is horrible, but fuzzies respect the authority of numbers.) Don't cheat—use real metrics. You want the problem to seem horrible to prove that the current rules are unsafe and need to change.
- Next, describe what conditions the solution must satisfy to keep the project and team safe—for instance, a rollback strategy, conditional compilation, policy settings, regular reviews, or manager approval. Don't just make this up; think about people's concerns. You must do this before you describe your solution or else you'll get battered by every fuzzy's apprehension. Remember, change has feelings associated with it.
- Now, describe your solution. Talk about how it meets the safety conditions (which you've previously worked out). Then talk about how it leaves the project in better shape. Suggest statistics that will demonstrate the results (like a percentage drop in regressions or pri 1 bugs or shorter stabilization times). Remember to keep it high level and simple. The statistics are particularly important because there's no other authoritative way to prove you've improved. To avoid the statistics being gamed, always use team instead of individual measures. Be sure to have a celebration when your new rules allow you to meet your goals.

Better together

It's easy to be cynical about liberal arts majors, or about anyone who thinks or works differently than you do. But different approaches bring out different values. Ultimately, we all benefit. It's mushy, but true.

By learning to appreciate the balance that fuzzies bring to our techie world, you can become far more effective. Understand your differences, adjust your approach, tune your message, and respect your audience. In the end, you'll have everything to gain and nothing to lose.

Eric Aside Since I left Microsoft's Engineering Learning and Development organization, my wife has been editing my columns for me each month. Even though we're quite different as people, we still make a great match. I can honestly say after 26 years together and 22 years of marriage that it's never been better.

November 1, 2005: "Undisciplined—What's so special about specialization?"

Why do we have testers? Why do we have PMs? Why do we have different disciplines at all? Isn't it wildly inefficient? Why don't we just have engineers who do whatever is necessary to ship quality products that delight our customers? Are programmers incapable of understanding customer needs or executing tests to verify their work? Of course not.

Eric Aside My introduction to this chapter talks about how important it is to have a team of people with a diversity of skills working together on a project. That doesn't mean those people need to be specialized into different disciplines with mutually exclusive responsibilities.

Yet, we somehow perpetuate these dysfunctional disciplines that create barriers, miscommunication, and conflict—also known as dependency death; handoff hell; black holes of blame; suffocating, senseless signoffs; and monotonous, mind-numbing meetings. Are we so insecure that everyone has to be "special"?

It's not just ridiculous, it's counterproductive. Devs and testers wait for PM specs, PMs and testers wait for code, and PMs and devs wait for test runs. Sure, there is always other work to do; but if everyone chips in on everything, then everyone is focused on the top priorities, everyone is a customer advocate, and everyone contributes to quality. It's the team way. It's the agile way. It's old school and it's new school. It's nirvana. What's not to like?

Days of future past

Microsoft didn't always have PMs and testers. We started out in nirvana and yet we strayed from paradise. Why? I've talked to old timers about it. Basically, not everyone wanted to do certain tasks, some people were better than others at certain tasks, and some important tasks weren't getting covered. There was a need, and people started to specialize and fill gaps.

Of course, that was long ago. The market was different, our situation was different, and we were less experienced. But if we went back to everyone doing everything, would history repeat itself, and would people specialize once more? Is there a fundamental principle at work?

Unfortunately, yeah, there is. Specialization is unavoidable and essential.

Before you get too excited one way or another, let me make two points:

1. The need for specialization is subtle; it doesn't always apply.
2. I'm right.

Take it to the limit

How do I know I'm right? Because of a great trick I learned on my high school math team. (Okay, I'm a dork. Let's move on.) Here's the trick: when you need to understand a broad problem, consider the extremes. In our case, the problem is optimal software development role structure. The extremes are coding an interview question and coding Windows Server.

When coding an interview question, you can do everything yourself. You get to know the customer (the interviewer), understand the requirements, spec the solution, code it, test it, and ship it. If you can't, you don't get hired. The moral of the story is that at the simple extreme, there's no need for specialization.

Coding Windows Server requires many more developers. But do some need to specialize? Absolutely, there are parts of the code that are way too complicated for more than a handful of devs to understand. But do you also have to specialize in customer design and testing (either quality assurance or control)? Absolutely, but that's harder to see because the scope of Windows Server is so big. Allow me to present far smaller example: an Xbox football game.

Football is a science

Coding an Xbox football game requires more than one developer. Again, specialization could be essential because of the computer graphics components, the AI components, the statistical components, and so on. But let's skip to the PM and test areas. Do you need specialists there? Absolutely, but not at the tiny feature level.

Writing a football game requires you to know every detail about football: every rule, every play, every formation, every team, every stadium, every player and salary, and on and on. However, not everyone on the dev team needs to know all those things. In fact, most don't need to know any of them. They've got other things to worry about, like computer graphics and AI. But someone needs to know—the game designer.

Someone needs to verify the results of the game. This person must be an expert video game player and football player who knows everything about how the game could be played and should be played. The complexity level is enormous, but it's at a different level of detail than the implementation. The developers who write the code operate at a much lower level of detail. Even the dev architect deals with a different slice of the complexity. But someone must verify the results at the user level—the game tester.

Windows Server is far more complex than an Xbox football game. There are hundreds, if not thousands, of different experiences that must be completely understood, designed, and tested at the customer's level of abstraction. I'd love to say that our engineers have big enough brains to keep all the details straight at every level of abstraction, and that they become refined experts about multiple, entire business models, module decomposition, and detailed implementation, but that's flat out absurd. The moral of the story is that at the complex extreme, specialization is unavoidable and essential.

Eric Aside This is the part that's lost on many Agile practitioners. Probably because there aren't many Windows-scale products that make the case for specialization so clear cut. Again, I return to the knowledge I gained that projects need to be managed differently at different levels of scale and abstraction. What goes for the hundreds or thousands of engineers at the product level doesn't make sense at the five-to-eight-engineer feature level.

The space between

So, what have we learned? At the simple extreme, specialization is superfluous. At the complex extreme, specialization is essential. Thus, the question isn't, "Should we have specialization?" The question is, "At what level of complexity or abstraction do we need specialization?"

I claim that specialization is a waste of time at the detailed feature level. Devs should understand the requirements of the detailed feature (the spec, scenarios, and personas), then design it, unit test it, and code it. Testers and PMs stay out of specing and testing at the detailed feature level. If devs aren't doing their jobs at that level of detail, then they aren't doing their jobs.

I further claim that specialization is essential at the product level. We need PMs to really understand the customers inside-out, arrange communication between them and the team, and keep the team focused on the ball. We need testers to ensure the customer will be delighted. Not that the product just works, but that it works the way the customer needs and wants it to work.

Stuck in the middle with you

The big argument, then, is where to draw the line in between? When do you stop needing the separate disciplines? Can devs design and test dialogs or APIs? How detailed do you need to go before PMs and testers are overkill and counterproductive? Personally, I think it depends.

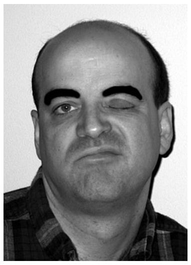
Where to draw the line depends on the product. If it's a familiar product with an intuitive purpose, then you probably don't need many specialists. If it's an unusual product with an obscure purpose, at least to engineers, then you need more specialists. Complexity naturally adds to the burden and requires more specialty.

In the end, you should avoid specialization whenever possible. It adds a tremendous burden and drives dysfunction. If you have jobs anyone can do, like check the build or fetch pizza, then everyone should help do them. Anything less is selfish, egotistical, and unprofessional.

When you do need specialists, accept and embrace them. Give them the opportunity to learn, grow, and lead. Our success depends on it. Just don't let it go to their heads. Work to avoid the pitfalls and barriers by using shared spaces, constant communication, and a focus on teamwork and team ownership. Remember: the team comes first, and pizza doesn't fetch itself.

Eric Aside I talk about the impact of inappropriately combining development and test teams later in this chapter in "Test don't get no respect."

January 1, 2009: "Sustained engineering idiocy"



Plumbing channels waste water into a series of larger and larger pipes till it is expelled. That's because sewage flows downstream, which explains the quality of goods that test, operations, and sustained engineering teams receive. After all, they are downstream of design and development.

I've written about pushing quality upstream for testers in "Feeling testy—The role of testers" (early in this chapter) and making services resilient to failure for operations using the five Rs in "Crash dummies: Resilience" (see Chapter 5.) Like most engineers, I've neglected sustained engineering (SE), also known as the engineering sewage treatment center. No, on second thought, that analogy implies that what we release to customers has been cleansed. SE is more akin to environmental cleanup after an oil spill—thankless, difficult, and messy.

Imagine what must go through the minds of those cleanup crews as they wash oil from the feathers of seabirds. Naturally, there's empathy for the birds (customers). There's frustration at the inevitability of mistakes that lead to tragedy (buggy software). And there's a palpable desire to have the jackasses who caused the spill be forced to take your place (the engineers who let the bugs slip by).

Eric Aside You take risks writing a controversial opinion column, and people can get hurt. That was certainly the case this time. Although I had three SE team managers review this column in advance to catch inaccuracies and unintended slights, many members of SE teams were deeply offended by it. Reading it again, I can easily see why. Not only did I question how SE folks should work, but I compared their jobs to sewage treatment and disaster recovery.

As obvious as those slights seem in hindsight, they weren't caught. Perhaps my reviewers knew that I was a big supporter of SE and wouldn't dream of insulting the engineers involved. My intent was to empathize with how SE teams are put into a thankless and difficult situation, not to devalue them myself. Unfortunately, good intentions don't reverse the harm done.

Here's part of what I wrote to an SE director who had thoughtfully expressed his concerns: "Thank you for pointing out this embarrassing, unintended, and damaging oversight on my part. Please know that I have nothing but admiration for your team and SE teams in general. It is my hope that by holding the core engineering teams more accountable for the quality of their designs and work that your jobs and our customers will greatly benefit."

You make the call

Should the engineers who design, construct, and test the code be the same engineers who fix the bugs found after release? This is the quintessential question of SE.

If the engineers who built the release fix the post-release bugs, you typically get better fixes, the engineers feel the pain of their mistakes, and the fixes can be integrated into the next release. Then again, the next release may not happen because its engineers are being randomized.

If you have a dedicated SE team you build up knowledge of the code base outside the core engineering team, you can potentially pay a vendor to sustain old releases, and you don't distract or jeopardize progress on new releases. Then again, SE teams get little love, their fixes can be misinformed, you duplicate effort, and the core engineering team isn't held accountable for their mistakes.

Tough call, huh? Nope, not at all. While both models can work, having the engineers who build the release also fix post-release bugs is far better. Only idiots believe a lack of accountability leads to long-term efficiency and high quality. Of course, the world is full of idiots, but I digress.

Someone's got to take responsibility

Yes, a dedicated SE team can work, but long term it will only cause grief for team members and customers. Why? Because you can mitigate post-release fixes distracting the core team, but you can't mitigate the problems with a dedicated SE team.

Let's go through those dedicated SE team problems again.

- **Little love** What would it take for the dedicated SE team to be appreciated as much as the core engineering team? A disaster, right? And what would it take on a day-to-day basis? Nonstop disasters. In other words, the conditions for loving the SE team are undesirable.
- **Misinformed fixes** To get a fix right, recognizing all the implications of changes, you need to deeply understand the impacted portion of the code base. Let's fantasize that the core engineering team has that level of depth. The core team is always considerably larger than the SE team. The SE team has no hope of truly appreciating the impact of fixes. Reality is only worse. Sure you can have the SE team consult with the core team, but doing that all the time defeats the purpose.
- **Duplicate effort** Whenever you have two teams fixing issues in the same code you duplicate effort, by construction. You've got two teams learning the same code, debugging the same code, changing the same code, and testing the same code. There's no getting around it, unless you neglect to incorporate the fixes into the next release, which is even worse.
- **Accountability for mistakes** The whole point of the dedicated SE team is to avoid derailing the core engineering team, protecting them from dealing with fixes. The core team doesn't correct its mistakes in the old code and doesn't know how to prevent those mistakes from recurring in the new code. What's worse is that there's no reinforcement of good and bad behavior. Conscientious heroes don't get to write more quality code, while careless villains fix past mistakes. Thus, we can never expect to improve. A great recipe for joyful competitors and sorrowful customers.

What do I do now?

In contrast, there's plenty you can do to avoid jeopardizing future releases while the core engineering team fixes prior mistakes. Let's run through the relentless, randomizing requests and resolve them.

- **Triviality** How do you avoid wasting the core team's time with issues that aren't software bugs or have trivial workarounds? You have a small dedicated team triage the issues. Note that this team isn't a development team. It's purely an evaluation team that determines which issues are worth fixing. That way, only worthwhile work is passed onto the core team.

- **Prioritization** How do you balance bugs fixes for the last release with work on the new release? You have the dedicated evaluation team prioritize the fixes. There are four buckets: immediate fix (the rare "call the VP now" issue); urgent fix (next scheduled update); clear fix (next service pack or update); and don't fix. These buckets send clear signals to the core team about which bugs to fix at what time.
- **Unpredictability** How do you make inherently unpredictable post-release issues easy for the core team to schedule around? You make them regular events. Deploy one update per month. The urgent fixes each month are queued up by the evaluation team. The core team sets aside the necessary time each month and the fixes are designed, implemented, tested, and deployed on a predictable schedule. This is just as good for customers as it is for the core engineering team. Everyone likes predictability.

In addition, the evaluation team can create virtual images for easy debugging by the core team, improve the update experience for customers, and reflect customer needs and long-term sustainability features back into future releases.

Eric Aside Many readers thought I was advocating elimination of SE teams. In fact, I wanted to keep SE teams but reduce their size and have them focus on issue evaluation and prioritization. These smaller SE teams also need to provide the orchestration and system support necessary to make SE run smoothly. They should design the right customer-centric fixes for the critical issues they find. They should prioritize those issues appropriately and drive for their customer resolution. However, the smaller SE teams shouldn't be the people fixing the bugs. That responsibility should go to core engineering team members so that they completely understand the issues they create and will know to avoid them going forward.

This won't hurt a bit

See, it's not that complicated. You save on staff. You get better fixes. You catch similar issues in advance. You achieve predictability. And you ensure the core engineering team is accountable for quality and learns from its mistakes. All it costs is a relatively tiny dedicated team to manage the monthly update process by evaluating and prioritizing issues. Even that team feels valued as a result of its differentiated and important role and its direct engagement with solving customer problems.

Yes, sewage flows downstream and no one likes cleaning it up. However, by putting some simple processes in place, you can reduce the sewage and have those responsible mop up the mess. To me that smells like justice.

Eric Aside What do you do if you are stuck on a dedicated SE team and are experiencing little love, misinformed fixes, duplicate effort, and no accountability from the core team? Here are a few ideas:

- Create a rotational program with the core team. Everyone spends a month or two a year on the SE team. It's not ideal, but I've already established that point.
- Measure your efficiency and effectiveness, perhaps by the average time to resolve issues for each bucket, the regression rate, team morale, and customer acceptance of fixes (a balanced scorecard). Optimize, publish your results, and show the core engineering team how great work gets done.
- You ship updates once a month—celebrate once a month.

May 1, 2011: "Test don't get no respect"

I love Microsoft. We've been together happily for many years. If you've been in a healthy long-term relationship, then you know what this means—there are things about Microsoft that make me curse, stomp, and spit. I've learned to tolerate them, but they still make me cringe.

A prime example is our disrespect for critical disciplines like testing. The test discipline is one of the two largest engineering disciplines at Microsoft and one of three key engineering triad disciplines. How can we not grant testers the same respect and opportunities we give the other two engineering triad disciplines—program managers (PMs) and developers? Perhaps our history provides the answer.

Microsoft was started by developers and run by developers for years. We're now run by a former program manager of sorts, so PMs receive begrudging respect. Developers can't draw and have no sense of style, so designers are becoming appreciated. Developers hate writing, so content publishing is at least considered necessary. Localization and media are magical things that just appear in the build. But test? Developers believe testing is easy, if not remedial, so developers think testers are beneath them.

Eric Aside Please note that I'm not giving any of these disciplines proper credit for all the work they do. I'm merely listing the superficial ways they are often viewed by developers.

It's a different kind of flying altogether

Since developers think testing is easy compared to development, they think they can do a tester's job. After all, isn't that the cool, agile way? Aren't we all just software engineers? Yeah, and everyone would get along if we just gave peace a chance. Don't be naïve.

Developers can verify that their individual components work as specified in isolation (unit testing). They have much more trouble verifying that their components work as a system, outside of isolation and spec. Why? Tunnel vision. Developers design and write their code for a certain purpose. That's the way developers think about it—as they should.

Real-world testing must verify that the code works appropriately when it's used in ways that defy all logic and purpose. To test code properly, you need to completely forget how and why it was written and instead think about how it might be used in arbitrary and even insidious ways. Developers don't think that way—they can't think that way and still develop with purpose. Testers do think that way. That's why we need testers.

That's easy!

Even though the test discipline is essential for high-quality software, some groups still consider converting all their testers to developers, expecting the combined development team to write all tests—it's a great way to get more developer headcount! The teams that actually go through with this change experience the following problems:

- **They lose their test leads and leaders** The former development leads tend to lead the combined engineering teams. (I talk about why in the next section.) This relegates the former test leads to individual contributor roles (ICs), and they usually don't take this reduction in responsibility well and leave. The former top test ICs also tend to leave—after all, clearly their skill set is not appreciated, no matter what sparkly purple lipstick the management team might put on this pathetic pig of a plan.
- **They lose their testers** After the test leads and leaders go, the former test ICs gravitate toward a development mindset. They gravitate there because testers get no respect, they are calibrated against developers even if they continue to work differently as testers, and they can see there's no career growth in test. After all, their role models left the team. Now the only path up is through development.
- **Team morale drops, especially among the testers** Losing team members and team leaders impacts morale. The most impacted are the testers still clinging to their discipline, who lose their role models, their self-identity, and their solid reviews. (Even great apples get bad reviews in an orange grove.)
- **Their code quality is initially higher and then gradually drops** The quality improves initially because developers suddenly realize they've lost their safety net and actually start writing unit tests, doing design and code reviews, and paying attention to those build warnings. (Of course, developers should have been doing this all along.) However, after a while, the system and boundary errors start creeping in and building up. No one is looking for them, so they are discovered by the wrong kind of testers—customers.

Combining development and test makes sense at the unit level (such as Test-Driven Development). This practice can also work at the component level for well-understood and well-factored components on teams that also have strong QA. However, combining development and test doesn't make sense at the system level. I talked about this at length in "Undisciplined—What's so special about specialization?" earlier in this chapter.

I just can't get enough

The lack of respect for testers is most apparent in leveling and career development. Depending on where they work in the company, testers are ranked one to three levels below developers and PMs. In other words, the test counterpart to a PM and developer—who are all working on the same project with the same scope—will tend to be a full career stage below his or her peers. Unbelievable!

"Yeah, but testing isn't as difficult as development and program management," says the dim developer. Really? Try it sometime. Try writing automation that works every time, even as developers alter configurations and data. Try performing penetration testing that closes gaps that foil sophisticated hacks. Try producing injection testing that discovers failure modes, system testing that finds sneak conditions, or end-to-end scenario testing that validates the billion-dollar bets we make. And I'm just scratching the surface.

"Yeah, but our test team isn't that advanced—they don't do all those things." Exactly! We don't value the test discipline enough to advance our testing capability to the same levels as PM and development.

Testing is frigging hard! We need great people to do it well, and then we need to pay, develop, and expect them to be world class. Our treatment of the test discipline is astonishing and pathetic. Believe or not, it used to be even worse, until senior test leaders from across the company started tracking the numbers and driving promotions.

Yes, it's true that testers do different jobs from PMs and developers. But we can't design and construct complex systems and think that testing those systems will be any less complex. You get what you pay for, and by spending less on testing we create imbalance. We sacrifice quality, productivity, and efficiency as a result. The sacrifice in quality is obvious. The sacrifice in productivity and efficiency comes from incomplete and fragile testing that results in higher error rates, more rework, and higher support and sustained engineering costs.

Eric Aside The refrain I hear when I complain about our commitment to testing is, "Yeah, but Google and Amazon have far fewer testers." Amazon carefully tests the systems that matter, like billing and account management. Google does endless analysis of its search results.

As for Microsoft, we are in a broader business than Amazon or Google. We are a platform company and an enterprise company. Our customers expect more from us. Quality is a differentiator and a necessity. Yes, products and services that aren't critical or sensitive and are updated monthly don't require the same in-depth testing as mission-critical products and services that span years between releases. While we don't need perfection in all things, we do need the right quality in the right products.

The world just wants us to fit in

It's bad enough to give up quality, productivity, and efficiency by expecting less from our testers than we do from their PM and developer counterparts. What's worse is that lowering expectations sends a clear message that to get ahead, a tester needs to become something else. This philosophy is nothing short of irresponsible and tragic.

- Lowering expectations for testers is irresponsible because some of our most challenging engineering problems involve testing—testing in production; testing many-core, highly parallel systems; testing one thousand plus machine services; testing globally distributed cloud services; testing secure and private cloud systems; testing hybrid procedural and functional languages; testing natural user interfaces; and on and on.
- Lowering expectations for testers is tragic because we send a message that testing is not a legitimate career path, when in fact it is a robust career path to the highest engineering stages for both test managers and test ICs. Instead of following this path, testers abandon their discipline to do something else, often with mixed results.

By expecting less from our testers, we are encouraging them to move away from a career they love—one that is essential to Microsoft's success and offers tremendous opportunity—and toward a career not of their choice that may inhibit their growth. It is a travesty.

Eric Aside Microsoft recently announced changes to its compensation plans that increase the base salary for most engineers, including testers. That's wonderful and I'm grateful. Investing in our test discipline is a separate matter. It is choosing to be just as sophisticated in testing as we are in program management and development.

Executives might reasonably ask, "Why would we increase our spending in testing? What's the return on investment?" While I believe a strong financial return exists, I'd turn that statement around. If we want to save money, why don't we decrease how much we pay developers by a couple of levels? Because our quality and innovation would suffer. Why would they suffer? Our products are platforms with wide and varied usage that require sophisticated engineering to orchestrate and improve—and test.

Tell me what it means to me

At a time when we should be investing in test, we continue to demean the discipline. We have great leadership at the highest levels within the test discipline, but far too few testers join these ranks each year. Even though test is far behind PM and development, they receive a smaller promotion budget. (The promotion budget is equal in proportion to the other disciplines, but that proportion is a distortion because of the higher salaries in PM and development.) Only a handful of test ICs and test managers reach the principal and partner stages to serve as role models.

How can we allow a critical and central engineering discipline to be so disrespected and damaged? Are we that vain or foolish to think testers aren't really needed or the problems aren't really that difficult?

Eric Aside As you might imagine, this column struck a nerve at Microsoft. Leaders from across the company told me how polarizing this was on their teams. People either loved it or hated it. The reasons varied and covered every aspect. I'm pleased with the open dialogue, and I'm pleased with all the stories people passed on about changes they've made or are making to improve the situation. However, we've got a long way to go as a company and industry.

It's time we put our money where our priorities are and push testing to the next stage. We hire the best—let's challenge them accordingly. Let's lay out the test career path all the way through to vice president and technical fellow. Let's start aggressively recognizing the talent we have and developing the talent we need. Testing deserves our respect—our customers, our partners, and our business depend upon it.

Eric Aside What can you do to help if you aren't a test or multidisciplinary leader?

- ❏ Accept and appreciate that the test mindset and skillset are different from development, yet their problems are equally complex and critical to our product quality.
- ❏ Write high-quality code from the beginning by using design and code reviews, code analysis (like PREfast and FxCop), and thorough unit testing, all of which allow testers to focus on their unique value of providing quality assurance and exposing system issues developers wouldn't normally detect.
- ❏ Encourage your test team to hire great full-time testers who focus on the truly challenging test problems we face—problems that when solved will improve the quality of our products, our testing, and our testers.

BTW, I could write a similar column on service engineers, who are even less understood than testers.

Chapter 5

Software Quality—More Than a Dream

In this chapter:

Some people mock software development, saying if buildings were built like software, the first woodpecker would destroy civilization. That's quite funny, or disturbing, but regardless it's misguided. Early buildings lacked foundations. Early cars broke down incessantly. Early TVs required constant fiddling to work properly. Software is no different.

At first, Microsoft wrote software for early adopters, people comfortable replacing PC boards. Back then, time to market won over quality, because early adopters could work around issues, but they couldn't slow the clock. Shipping fastest meant coding quickly and then fixing just enough to make it work.

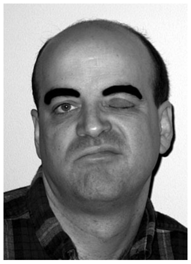
Now our market is consumers and the enterprise, who value quality over the hassles of experimentation. The market change was gradual, so Microsoft's initial response was simply to fix more bugs. Soon bug fixing was taking longer than coding, an incredibly slow process. The fastest way to ship high quality is to trap errors early, coding it right the first time and minimizing rework. Microsoft has been shifting to this quality upstream approach over the time I've been writing these columns. The first major jolt that drove the company-wide change was a series of Internet virus attacks in late 2001.

In this chapter, I. M. Wright preaches quality to the engineering masses. The first column evaluates security issues. The second analyzes why quality is essential and how you get it. The third column explains an engineering approach to software that dramatically reduces defects. The fourth talks about design and code inspections. The fifth describes metrics that can predict quality issues before customers *experience them. The sixth focuses on techniques to make software resilient. And the chapter aptly finishes by emphasizing the five basics of software quality.*

While all these columns provide an interesting perspective, the second one, "Where's the beef? Why we need quality" stands out as an important turning point. When I wrote it few inside or outside Microsoft believed we were serious about quality. Years later, many of the concepts are taken for granted. It took far more than an opinion piece to drive that change, but it's nice to call for action and have people respond.

—Eric

March 1, 2002: "Are you secure about your security?"



I know security is serious sh*t. I know that every time some hacker half-wit exploits a small code or configuration imperfection in a system that he could never dream of writing himself, the dung beetles in the press will feed off this excrement with gleeful abandon, telling our customers that our code stinks—simply because some pockmarked malicious pipsqueak managed to manipulate two lines of code, out of a million, into an illegal perversion. I know this.

Microsoft has millions of lines of legacy code, huge farms of networked servers, and hundreds if not thousands of external partners and dependencies. Each of these has the potential to be the next victim of vile, vindictive, vacuous, vessels of vomit whose mothers still wash their clothes. But which items should we focus on securing first?

Eric Aside Ah, the good old days when hackers were just misguided youth trying to make a name for themselves or fighting the powers that be. These days, hacking is big business—either preventing it, tracking it, or engaging in it for organized crime. In retrospect, my anger at the early hackers was misplaced. The world isn't a place where you can just hope everyone plays nicely. The early hacker wake-up call set us on the proper path of writing secure solid code.

Beware the swinging pendulum

Some people say that every possible vulnerability must be corrected. That's commendable—but crazy. We can't get so paranoid that our products become unusable.

When I worked at Boeing, people assigned to "black projects" worked in buildings with no windows, were disconnected from the network, and every night removed their disk drives and locked them in a vault. With all these protective measures, these projects were still considered vulnerable.

Even the biggest security hawks probably don't think our customers should be required to blacken their windows, stay off the web, and remove their disk drives. However, we must raise the security bar far higher than in the past and require accountability for it from top to bottom in our organizations. The key is to remember that this is all about delivering a trustworthy and *delightful* experience to our customers.

Other people say that we only need to focus on securing our firewalls, protocols, and common language runtime. They assume that if these are secure, we have nothing to worry about. This is ignorant and downright dangerous thinking.

Writing Secure Code (Microsoft Press, 2002) by Michael Howard and David LeBlanc (a must read) has an entire chapter on writing secure .NET code, which references almost all of the other 15 chapters. This chapter explains that vulnerabilities are not limited to buffer overruns, insecure protocols, and unguarded ports. Even if these items were the only attack points, firewalls, secure protocols, and managed code wouldn't be enough protection from malicious data.

Eric Aside I refer to the first version of *Writing Secure Code* in this column. The column was written shortly after the first version was published. Naturally, later versions of the book have even more useful information. Also, Michael Howard has a great Kiwi accent.

Do the right thing

Of course, the right thing to do is to consider every possible vulnerability and rank each in terms of the risk to the customer. Finding vulnerabilities is not as hard as it sounds. By breaking your web or client application into components and interfaces, you can quickly spot potential issues and classify them with the STRIDE model. By searching your code for dangerous APIs (Appendix A in *Writing Secure Code*), properly restricting permissions, and checking inputs, you can find a ton of easy pickings—the kind that hackers look for. Although a deeper evaluation is necessary to catch more subtle issues, these simple steps will give you a great head start.

Eric Aside STRIDE is a mnemonic device to help people remember the different kinds of security threats: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege. *Writing Secure Code* has all the details.

Next you have to assess the risk of each vulnerability. A vulnerability that allows hackers to discover how we keyword clip art is less critical than one that allows hackers to discover a customer's private data. There is a higher probability that a vulnerability will be exploited if you can copy a hack from a website and change a couple of values in the script, as compared to a hack that must be tuned to each instance, requires detailed knowledge of the code, and must be written in a low-level language. The more critical the effects, the higher the risk; the lower the probability of exploitation, the lower the risk. Divide the criticality by the chance and you've got your risk assessment. (Table 2-2 in *Writing Secure Code* details this process.)

You're only as secure as your weakest link

Computing security risks sounds simple enough, but how do you calibrate acceptable risk across a big product like Windows? First, your team has to agree on definitions of the criticality and chance ratings. The commonly suggested range for each is from one to ten, where a criticality of ten is highly critical and a chance of ten is highly unlikely.

Next, calibrate these ranges according to your group's agreed-upon sense of importance. Is getting read-only database access to product catalog data a criticality rating of two or eight? What about the same access to user data? If you need to write custom COM code to exploit a vulnerability, is it assigned a chance rating of four or nine? What if that code can be written in VBScript? (One very helpful standard for risk assessment of interfaces can be found in Table 14-1 in *Writing Secure Code*.)

Eric Aside These days, Microsoft security experts have written custom templates in Team Foundation Server (TFS) that automatically map vulnerability types to priority and severity values for security bugs. These custom TFS templates are used internally by big teams to further normalize and simplify the bug tracking process.

After your group has standard definitions of criticality and chance, a high-level triage or war team can set a balanced risk bar across the organization. We already use this process to settle differences between component teams working on large products and to set a consistent quality bar. The same can and should be done for security issues.

Eric Aside To differentiate low-level product and feature triage from high-level product-line triage, we have a variety of funny internal names: war room, box triage, and ÿber-triage, to name a few. Personally, I could do without the war references.

Lead, follow, or get out of the way

So are you on board with responding to the security challenge? Do you hate the idea of a group of arrogant, asocial peons pushing us around?

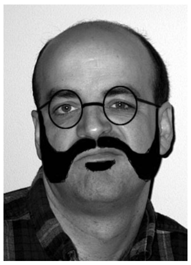
Perhaps you are thinking that these loser lawbreakers should be kept off the Net, and that pandering alarmist reporters and editors should have a sense of decency, fairness, and responsibility, not make heroes of hackers and not vilify Windows beyond all other platforms and systems. Perhaps you are thinking that if these hacking incidents were taken with the proper perspective, then maybe we wouldn't all be going through this fire drill. Unfortunately, this is the world we live in, fair or unfair, true or exaggerated.

Let's face it. No one likes being hacked and few of us praise hackers for their creativity and civic service. But letting snidely, sniffling, scurvy scum have control over the hearts, minds, and computers of our customers is completely shameful.

Sure Linux, Oracle, Sun, IBM, and AOL have as many, if not more, security problems than we do. But as BrianV says, "We are the leader in this industry and we have to lead!" Nothing short of a full commitment is acceptable.

Eric Aside Brian Valentine (BrianV) was the senior vice president of the Windows division at the time. Nearly a decade later, security is baked into all our products and services. It is a way of life we all accept, appreciate, and strive to understand.

November 1, 2002: "Where's the beef? Why we need quality"



This month, *Interface* focuses on the lessons that we learned from security pushes around the company. What about the lessons that we haven't learned? What about the dumb things we are still doing?

Eric Aside *Interface* is the name of the Microsoft monthly internal webzine in which my opinion columns first appeared. The webzine published its last issue in February 2003.

The security fire drill exposed more than security holes in our software. It further exposed the shoddiness of our work and left many folks wondering what the next fire drill will be. Guesses include: privacy, availability, supportability. How about quality? Anyone heard of quality? What the heck happened to quality???

Check-in time on most dev teams is like amateur hour. The kind of garbage that passes as a first cut is pathetic, and fixing it later is like signing our own death certificate. Why? Because we won't fix problems before release that are too superficial, too complex, or too obscure. We won't fix bugs before release that are found too late or simply not discovered until after we ship.

So what, right? We've been shipping this way for years. What is true this year that wasn't before? Oh my, where to begin…

Eric Aside Much of what I wrote about here nine years ago has changed. That doesn't mean we are yet where we want to be. However, we've radically increased the amount of unit testing, automated testing, code review, and code analysis we perform both before code gets checked into the main source tree and before we ship. These days we can actually use weekly builds for mission-critical internal tasks and day-to-day work, and daily builds to make incremental improvements to production services.

Things have changed

First of all, today we are trying to sell into markets that require turnkey solutions—that is, you turn the key and it works. These markets require turnkey solutions because the associated customers do not have the expertise to work around the problems. So if it doesn't work right away, we have to fix it right away.

We have entered two major turnkey markets: consumer products and the enterprise. If you're smart, you're wondering how our competitors have succeeded in these markets.

For the consumer market, our competitors have kept their products small and simple. That way there aren't many failure modes; and if they do fail, the product can quickly restart and recover. We are selling products with far more complexity, and functionality. However, this means we have more failure modes. To stay competitive, our products need to be better with fewer failures and they need to restart and recover faster.

For the enterprise market, our competitors have supplied armies of support personnel and consultants. For many competitors, this is the biggest part of their business—they actually make money on their complexity and failures. When their products collapse, our competitors immediately dispatch their own squadron of people to fix the problems and keep the enterprise up and running.

We don't follow this business model. We sell less expensive products in high volume and provide minimal support in an effort to maximize profits. However, this means that we can't afford to break as often and that we must quickly fix or recover from the few failures that we do have.

Eric Aside Our "minimal" support has expanded significantly as the Internet provides new models for support, but Microsoft is still a volume software and services provider.

Good enough isn't

The second way things have changed for us as a company is that our key products are now good enough. Actually, feature-wise our key products—Office and Windows—have been good enough for years.

Being good enough means that we’ve got all the features that our customers need, or at least those that they think they need. This hurts us two ways:

- People stop upgrading to the next version. After all, the current version has everything that they think they need and upgrading is painful and expensive.
- Any software copycat can create a viably competitive product by just referring to our widely distributed, online, fully automated specifications (the products themselves). If the copycat does a better job, making the software more reliable, smaller, and cheaper (say like Honda did to Chrysler), then we’ve got a big problem.

Think it can’t happen? It already has. (Does Linux ring a bell?) Linux didn’t copy Windows; it just ensured that it had all the good-enough features of a Windows server. Right now there are developers working on Windows-like shells and Office-like applications for Linux. Even if they fail, you can bet someone will eventually succeed in developing a superior product—as long as we leave the quality door open.

We can’t afford to play catch-up with our would-be competition. Detroit has been fighting that losing battle for years. We must step up and make our products great before others catch us.

The good news is that we have some time. Other commercial software companies big enough to copy Office or Windows are poorly run and are way behind us in the PM and test disciplines. The open-source folks lack the strategic focus and coordination that we have; they rely on a shotgun approach hoping to eventually hit their target. We can beat all competitors if we raise our quality bar—ensuring fewer failures, faster restart, and faster recovery—and if we focus on our key customer issues.

Hard choices

But as anyone can tell you, nothing comes for free. If we focus more on quality, something else has to give. At a high level, the only variables that we control are quality, dates, and features. For projects with fixed dates, quality means fewer features. For projects with fixed features, quality means adding time to the schedule.

Eric Aside Actually, I don’t completely believe this anymore. I’ve seen great efficiency gained by removing waste from the system (as you can see in “Lean: More than good pastrami” in Chapter 2) and fixing problems early. While it might not be enough to give the company summers off, I believe it is enough for high quality not to cost us features or time. Yes, it’s not as quick as the early days when the quality bar was low, but compared to our recent long stabilization periods, doing it right the first time is as fast if not faster.

Before you balk at this thought process, BillG has already made our choices clear in his article about trustworthy computing:

> *In the past, we've made our software and services more compelling for users by adding new features and functionality and by making our platform richly extensible. We've done a terrific job at that, but all those great features won't matter unless customers trust our software.*

The only question is: Are you going to follow through?

There are three principal areas to focus on to improve the quality of our products:

- Better design and code
- Better instrumentation and test
- Better supportability and recovery

Let's break them down one at a time.

Time enough at last

Few developers wouldn't love more time to think through their code and get it right the first time. The trouble is finding the time and having the self-discipline to use that time wisely. So, what would you do if you had more time? As a manager, I would spend more time with my people discussing design decisions and reviewing code.

Two key design issues I'd emphasize are simplicity and proper factoring:

- **Simplicity** Keeping the design simple and focused is key to reducing unintended results and complex failures.
- **Proper factoring** This helps keep each piece of the design simple and separable from the others. It also makes it easier to enforce a sole authority for data and operations, and to maintain and upgrade code.

Eric Aside Test-Driven Development (TDD) accomplishes both these results for implementation design. You can take a similar approach to TDD for component design as well, though the tests are sometimes no more than thought experiments.

I'd also give devs extra time by pairing them to work on each feature task. This serves to

- Double the time that each dev has to do the work because you schedule the same task length as if one dev were assigned.
- Allow for peer reviews of designs and code.
- Provide each feature with a backup dev in case the primary dev becomes unavailable.

To help my devs apply self-discipline, I'd

- Schedule completion dates for dev specs (also known as design docs and architecture docs).
- Make each backup dev as responsible for feature quality as the primary developer is.
- Measure regression rates and build-check failures to use as feedback on quality at check-in. (Sure, these measures are imperfect, but what did you want? Bugs per line of code?)

Eric Aside These days I'd use churn and complexity measures instead of regression rates. See "Bold predictions of quality" toward the end of this chapter for more.

Checking it twice

It's never enough to think that you have the code right; you've got to know. Check it from the inside with instrumentation and from the outside with unit tests. The more you do this, both in terms of coverage and depth, the better you can assure quality.

And, NO, this is not a tester's job. Test's job is to protect the customer from whatever you miss—despite your best efforts. The testing team is not a crutch on which you get to balance your two tons of lousy code. You should be embarrassed if test finds a bug. Every bug that they find is a bug that you missed, and it better not be because you were lazy, apathetic, or incompetent.

So how do you prevent bugs from ever reaching testers and customers?

- **Instrumentation** Asserts, Watson, test harness hooks, logging, tracing, and data validation can all be invaluable (even instrumental—sorry) in identifying and pinpointing problems before and after check-in.
- **Unit tests** Testing can often make the biggest difference between good and exceptional, between barely functional and solid. There are lots of different kinds of unit tests; you, your backup, and your peer in test should pick those that are most appropriate for your feature:
  - Positive unit tests exercise the code as intended and check for the right result.
  - Negative unit tests intentionally misuse the code and check for robustness and appropriate error handling.
  - Stress tests push the code to its limits hoping to crack open and expose subtle resource, timing, or reentrant errors.
  - Fault injection tests expose error-handling anomalies.

The more you verify, the more code paths you cover, the less likely it is that a customer will find fault in your work.

Physician, heal thyself

Even if you design and code your feature well, and even if you instrument and test it well, there will still be bugs. Usually those bugs involve complex interactions with other software. Sometimes that software isn't ours; sometimes it's old; and sometimes it isn't well designed or tested.

But just because bugs will always occur doesn't excuse you from making bugs rare, nor does it excuse you from taking responsibility when they do occur. You still have to help customers fix or recover from these problems, ideally without them ever noticing.

A wonderful example of recovery is the IIS process recycling technology. Any time an IIS server component dies, the process automatically and immediately restarts. The customer, at worst, only sees a temporary network hiccup repaired by a simple refresh.

Office XP also has a recovery system, though the user is made more aware. When an Office app fails, it backs up the data, reports the problem, and automatically restarts and recovers the data upon request. These solutions are not terribly complicated, but they offer a huge benefit to customers and save tons of money in support costs.

Eric Aside Read "Crash dummies: Resilience" later in the chapter for a far more in-depth discussion of resiliency.

If you can't get your product to automatically recover and restart, you should at least capture enough information to identify and reproduce the problem. That way, the support engineer can easily and quickly understand the issue and provide a fix if one already exists. If the issue does not have a known fix, the failure information can be captured and sent to your team so that you can reproduce it and design a fix for the customer right away.

Capturing enough information to identify and reproduce problems is not as hard as it sounds:

- Watson currently does a great job of identifying problems, and future versions of Watson will make it easier to reproduce those problems on campus.
- SQL Server does a great job of capturing a wide variety of customer data, which allows everything from precisely reproducing all the changes to a database to simply dumping the relevant state when a failure occurs.

Step by step

Okay, you follow all these suggestions. Was it worth it? Is the code really better? Did you miss something? If these security pushes have taught us anything, it's that problems aren't always obvious and that ad hoc methods and automated tools don't find all of our issues.

The pushes also reminded us that a couple of missed exploits can cost Microsoft and our customers millions of dollars and lead to a double-digit drop in our customers' perceptions of product quality.

What techniques can we borrow from the security folks? Two immediately come to mind:

- Decompose your product into components like you would for a threat model, and look for quality issues within each piece. Ask: How should we design each component? How can we make each component instrumented, testable, supportable, and recoverable? Applying more structured engineering methods to improving quality will yield more reliable and comprehensive results.
- Reduce the failure surface area of your product; that is, reduce the number of ways that your product can be misused. Cut options that allow customers to invent procedures that you didn't intend and that are included just because you thought someone somewhere might desire that option. Simplify each feature so that it performs the one task that you designed it for and that's all. Remember, unnecessary complexity just hides bugs.

Too much to ask?

By now you are surely thinking that I'm insane. There's no way your PM, test, and ops teams—let alone your managers—are going to give you the time to take on everything that I'm suggesting. Actually, it's not as bad as you might think. Many of these practices are tied to each other:

- Proper design welcomes testability and supportability in your products.
- Instrumentation helps identify and reproduce problems.
- Testing shows weaknesses where you need to recover.

To buy more time to do this right, make improving quality a win for the whole team. Define supportability and recovery as features. Get PM, test, and ops into the act.

When we build our products right and customers regain confidence in our work, we will leave our competitors in the dust. They can't match our features; they can't match our innovation and forward thinking; and if we show what great engineers we can be, they won't be able to compete with our quality.

Eric Aside "Where's the beef? Why we need quality" is one of my favorite columns. Nine years later I still get pumped reading it. Quality is a pursuit not a destination. Even though we still have far more to improve, I'm proud of how much we've done.

April 1, 2004: "A software odyssey—From craft to engineering"

Time to weigh in on a development debate for the ages. No, not where to put the curly braces (they go on their own lines). The debate is, "What the heck are developers?" Are we like designers and artists, creative people who need time to think and imagine? Is development a craft and we are craftspeople? Or, as our titles imply, are we "software engineers"? That last suggestion really bristles people. Get over it; the issue has been decided for us.

Oh, there are plenty of people who still think that this is an open debate. I've followed the developer aliases, seen the websites, and heard the arguments. Heck, I've made the arguments, claiming forcefully, "We're developers, not engineers." (See "Pushing the envelopes: Continued contention over dev schedules" in Chapter 1).

Developing software is a creative process. It's an unpredictable process, dealing with custom components that have poorly understood properties. Many believe that there's no hope of applying engineering practices to this process in our lifetimes. They contend that we are at worst hacks, cowboys, cowgirls, and reckless amateurs, and at best creative craftspeople. Well, treating software development as a craft is no longer adequate.

Craft a desk, engineer a car

Don't get me wrong, I love crafts. There's nothing like a sturdy hand-crafted table and chairs, an elegant hand-crafted timepiece, or even a well-designed and crafted home in which you can raise your family. I just don't want to drive my car over a well-crafted bridge. I don't want someone to stick a well-crafted pacemaker in my chest. And I don't want to rely on well-crafted software to run my business, protect my assets, or direct my actions. I want well-engineered software for these tasks and so do our customers.

So, what separates a hack from a craftsperson and a craftsperson from an engineer? A hack learns as he goes, acts then thinks, cleans up his mess after someone's stepped in it. Sound familiar? In contrast, a craftsperson studies, plans, uses the best practices and tools, and takes pride in her work. This describes the best of us who develop software. But craftsmanship doesn't quite reach engineering status because you still don't know what you are going to get. Craftsmanship lacks certainty and predictability; you make your best estimate instead of knowing.

It’s what you know

Engineering, on the other hand, is all about knowing instead of guessing. It’s about measuring, predicting, controlling, and optimizing. An engineer doesn’t wonder about things, he looks it up. An engineer doesn’t estimate, she calculates. An engineer doesn’t hope, he knows. This doesn’t mean that engineering lacks creativity or innovation, just that there are known boundaries to safe behavior that must be enforced to achieve reliable results.

But we all know that software is unpredictable. How can we possibly apply structured engineering practices to software? The secret is so obvious; it kills me that I didn’t see it sooner. Don’t try to predict the software, predict the people who make it. The constant in software development isn’t the software, it’s the developer. People are creatures of habit and our habits are predictable. That realization may not sound profound, but it changes everything.

To thine own self be true

It may hurt to think that you are predictable, but you are. A little introspection will reveal that truth. You make the same mistakes over and over again. You take about the same amount of time every time to write certain types of functions or objects. You even write them by using about the same amount of code. It’s scary but true, and more importantly, measurable and predictable. Holy sh-t.

Okay, so I didn’t believe it either at first, but then I spent a couple of weeks measuring myself programming and graphing the results. Like the other 4,500 programmers who tried this before me, I’m an open book. For any given type of function or class, I take roughly the same amount of time, write the same number of lines, make the same number of mistakes of the same kind, and take about the same amount of time to fix them based on type. This insight is embarrassing, but powerful.

Eric Aside The two weeks I spent measuring myself were part of a course on the Personal Software Process (PSP) from the Software Engineering Institute (SEI). PSP is part of an engineering team approach to software development called the Team Software Process (TSP). I was impressed with their demonstrated results and the theory behind them. My team tried TSP for a while. I’ll talk more about how it went shortly.

If you just draw a diagram of the classes that you need to write, you can know with measurably high confidence how long it will take, how many lines you’ll write, how many bugs you’ll have and of what type. You can also know how many bugs will surface in design review, code review, by the compiler, in unit test, and by the test team.

What's in a number

So what? So this: If you know how many bugs you'll find, you can know to keep looking, you can know when you've looked enough, you can know how many other people should look and what they should look for. You can say, with confidence, we've found and fixed 99.9999% of the bugs, period. In other words, you can know instead of guess. Congratulations, you're an engineer.

Okay, what's the cost? How much crud do I need to track in order to do high-confidence predictions? Here's the full list of the measurements you must collect:

- **Work time spent between checkpoints** This is the amount of time that you actually spent doing work between each checkpoint. (Checkpoint examples include design complete, design review complete, code written, code review complete, code builds cleanly, unit tests run clean, and code checked-in.)

  > **Eric Aside** If you use Kanban, which you should, this data is trivial to collect. You can derive the data directly from the Kanban board by using a cumulative flow diagram (a burn-up chart). It's simply the time each story takes to move through each Kanban bin.

- **Time and rework spent on check-pointed work** This is the amount of time spent on reworking stuff you "finished" in an earlier checkpoint, along with a one-line description of what happened and some categorization so that you can reference it later. (This is typically a task like design changes after design complete or code changes for bug fixes or whatever.)

  > **Eric Aside** This is also fairly easy to collect from a Kanban board if you track story rework separately from story linear flow. Corey Ladas covers this in his article "Accounting for bugs and rework" (*LeanSoftwareEngineering.com*).

- **Number of lines of code you added, deleted, or changed** This one is obvious and can be automated easily.

That's it. It's all information that you can get with a decent timer and notepad, although teams are working on tools to make it even easier. You must be consistent for accurate results; but with only these data points, you get more information than you could have dreamed possible.

For instance, you can answer questions like, "How much time did I spend doing real work this week?"; "How many times did we have to change the API, and how long did that take?"; "What percentage of bugs is found in code review?"; "How does the percentage of code review bugs found relate to the time spent in code review?"; "What kinds of bugs are mostly found early vs. late?"; "What kind of bugs take the most time to fix, when are they introduced, and when are they found?"

It's their habits that separate them

So, what's the catch? The data is only good for one person. Everyone's habits are different; you can't compare my data to yours and have a meaningful discussion. This is actually a good thing because managers shouldn't be using data for comparisons anyway. As I discussed in my article "More than a number—Productivity" (found in Chapter 9), when managers use data against people the measures get gamed.

Although data can't be shared or compared for individuals, it can be aggregated for teams. This is a manager's dream come true. You can do all this prediction and quality management on the team level with little effort and extremely high levels of accuracy. Because aggregated data drives toward the mean, the results for teams are no less accurate than they are for each individual. You could manage 100 people and be able to predict completion dates and bug counts to the level of accuracy of a single individual. Yowza!

Think big to get small

Okay, what's the punch line? Maybe engineering software is possible to some extent. Maybe you can predict code size, bug counts, development time, and so on starting from just a swag at the list of objects in your system. How does that data translate to results? Teams both inside and outside Microsoft that used data like this to target and control their bug counts have lowered their defect rates from the typical 40–100 bugs per thousand lines of code to 20–60 bugs per million lines of code. In other words, the typical Microsoft team of 15–20 devs that produces 3,000–5,000 test bugs per year would instead produce 3–5 test bugs per year.

And yes, those low bug rates include integration bugs. People bicker and moan about how complicated bugs can be in our big software systems. It's true; you do find a large percentage of bugs during integration. But what kind of bugs are they after you find them? Are they wildly complex timing bugs with weird unpredictable multithreaded interactions? Maybe one or two of them are, but the remaining thousands of bugs are brain-dead trivial parameter mix-ups, syntax errors, missing return value checks, or even more commonly, design errors that should have been found before a line of code was written.

Good to great

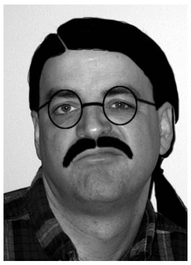
Of course, the way great teams control their bug counts is by using design reviews, code reviews, tools like PREfast, and unit testing. However, these methods alone only make a developer a craftsperson, dropping bugs by a factor of 10 or so, not by a factor of 1,000. The drop isn't that large because you have to guess where and how to apply your craft; you don't know. By investing in a small number of measurements and taking advantage of your own habits, you can know. You can graduate into being an engineer and earn that factor of 1,000 improvement.

That's a big and necessary step for delivering the quality and reliability that our customers demand. You must also discern the requirements and create a detailed design that meets them, but those amusing subjects will have to wait until next time. For now, dropping your bug counts to around 10 per year would be a nice start.

Eric Aside So how did my team's experiment with TSP go? Did we achieve a 1,000 times reduction in bugs (rework)? Not exactly. To be fair, my team isn't a typical team and we didn't stick with TSP long enough to get reliable results. The problem wasn't the methodology, though it was unnecessarily burdensome and rigid at times. The problem was the tools—they were unusable and unreliable, and they didn't scale to large teams.

Since that time, we've focused on process changes that are easier to adopt but that still make a huge difference. These include Scrum, TDD, planning poker and Delphi estimation, inspections, the basics of design and code reviews, unit testing, code analysis, and, most recently, the adoption of Kanban and scenario-focused engineering.

July 1, 2005: "Review this—Inspections"



When you get invited to a spec review meeting, what's the first thought that enters your mind? My guess is, "Necessary evil," or "Okay, time to see how bad things are." Perhaps you just let out a resigned groan like the guy who cleans up after elephants at a circus.

How about when you attend a code review? Ever leave feeling uneasy—like you just left your home for a week, but you can't remember if all the doors are locked and the lights are off?

In design reviews, do you just sit there while two people debate a few areas so long that your issues never get heard? Even worse, half the people haven't read the docs in advance, which may be a good thing because the author didn't run a spell check on his Swiss-cheese–excuse of a design.

Well sir or madam, you have a problem. Unfortunately, your team is suffering from a common condition: "Failing to Orchestrate Collective Knowledge Effectively for Design." You're FOCKED.

A bad combination

You get FOCKED when you don't differentiate three activities:

- Generating ideas and solutions
- Receiving feedback on work in progress
- Assessing quality and detecting issues in completed work

Most teams combine these activities into one—a review meeting. Some might call it a clustered version of FOCKED.

Naturally, this is a bad move. By combining three different goals into the same exercise, you not only guarantee failure in every activity, you frustrate and confuse the participants and send mixed messages to the organizer.

How do you avoid consenting to this treatment? Understand what your goals are, and choose the most appropriate and effective method to achieve them. Brainstorming, informal reviews, and inspections are the methods I suggest.

The perfect storm

When you want to generate ideas and solutions, use brainstorming. You can brainstorm in small or large groups or one-on-one in front of a whiteboard. With more than four or five people, use an affinity technique: Everyone gets a stack of Post-It notes. They write down as many ideas as they can and group the notes into common themes.

Regardless of how you brainstorm, the goal is to get as many ideas as possible. There's no bad suggestion. The best solution may result from parts of many initially unworkable proposals.

Brainstorming is great to use early in the design and specing process, before you've written much down. Get together with key stakeholders and feature team members and crank out ideas.

If you can't pick a solution, use the Pugh Concept Selection to decide. This method uses a table to rate each solution against each independent requirement. The rating is positive, negative, or zero, based on fit, as compared to your default choice. Then scale the rating based on significance. The solution with the highest total wins.

Online Materials Pugh Concept Selection (PughConceptSelectionExample.xls)

Better yet, don't pick a solution. Keep each design idea until you discover some constraint or factor that makes it unfeasible. Eventually, you'll be left with the one solution that optimally meets all your requirements. The Toyota folks call this "set-based design."

Who's in charge?

Be careful not to confuse brainstorming with "design by committee." Although many people may contribute and discuss different ideas, there should be one owner of the design. The owner has the final say; her name is the one on the design.

Having a single owner gives the design clarity and consistency. It creates accountability for meeting the requirements and a champion for understanding and defending the spirit of the design choices.

In contrast, design by committee is for groups of wimps who need others to blame when their spineless consensus crud implodes under the smallest of strains.

So, what do you think?

Sometimes you just need to know if you're headed in the right direction. Reviews of one-page specs can provide this. So can informal peer reviews through e-mail or at team meetings. Reviews of work in progress act as checkpoints against wasted effort caused by early naïve or subtle mistakes.

The problem is that people use this type of review under the guise of issue detection. Informal reviews are pathetic at issue detection compared to other methods. Yet almost all spec, design, and code reviews done in engineering groups are informal reviews.

Trying to use informal reviews for issue detection (like a quick, "Hey, have a look at this bug fix before I check it in") sets up you, your team, and our products for failure. You get a half-hearted review by unprepared reviewers. Your team gets a constant sense of, "Are these reviews really worth it? I feel like we miss stuff." And your products get buggy code that you still end up fixing later—hopefully before you ship.

Teams can get disenchanted or think that they have to work the reviews even harder. The shame of it is that informal reviews are wonderful for getting feedback. They were never meant for quality control or assurance. There's no reason to drag in the whole team, berate them for not reading the docs or code, then slog through an ineffective meeting, all for an informal review.

If you need feedback, ask for feedback. Get feedback early. Make it fun and casual, then thank folks for their help. Walkthroughs (guiding a group through your design or code) and informal reviews are great ways to do this. If you need issue detection for quality control and assurance, you need to use inspections.

It’s just a formality

Inspections are for issue detection in completed work. Period. They stink for generating ideas and solutions or for giving feedback. However, if you want to find all the bugs in your designs and code before it ships, you want inspections.

Inspections list all the issues found and give you an accurate estimate of all the issues not spotted. The secret is in the formality. Inspections take no more time than thorough informal reviews, but their formal procedure leaves no unaccounted issues.

Eric Aside The particular approach to formal inspections I describe here is based on the one used by the Software Engineering Institute’s Team Software Process (TSP).

Here’s a quick summary of the procedure:

- **Plan** Ensure that the work is complete, then schedule the meeting.
- **Overview** Give all your inspectors (that is, reviewers) enough context to understand your work, a checklist of issue types to find, and a copy of the work itself.
- **Prepare** Tell the inspectors to individually list all the issues they find (based on the issue-type checklist).
- **Meet** Get inspectors together to merge issue lists, agree on duplicates, determine which issues must be fixed, and decide if the work will require a re-inspection after the issues are corrected.
- **Rework** Address all the must-fix issues and as many minor issues as you choose.
- **Follow-up** Learn the root cause of the issues you fixed, and update your checklists to avoid those issues in the future. Repeat the inspection process as required.

The magic of inspections is found in the checklist of issue types and in the issue-list merge. However, before we get there, the work must be ready.

Are you ready, kids?

The point of inspections is to find all the remaining issues in your completed work. If the work is incomplete and full of gaps and basic errors, the inspectors will get bogged down in triviality, lose focus on the tougher checklist issues, and hate your guts for wasting their time.

Before you schedule the meeting, make sure you have a clean spec, design document, or source code. Ensure that specs and design docs cover everything you need to have verified. Ensure that source code compiles, builds, and functions. This includes having a clean PREfast run and completing any other aspects of your group’s initial quality bar. Ideally, your team should have someone assigned to verify that the work is complete before scheduling an inspection.

A great way to ensure your work is complete is to conduct a personal inspection. Lots of developers already do this. They look through their own code or designs trying to find issues. Add a one-page checklist of your common mistakes and you've got a personal inspection. You'll want your checklist to contain the most common ways you mess up. Every person's checklist is unique because everyone makes different kinds of mistakes; for example, I'm great with resource cleanup, but I constantly switch parameters around. Update your checklist after every inspection, removing issues you no longer find and adding new issues you start to notice.

Checking it twice

After your work is complete, send it to inspectors (a few days before the scheduled meeting). Attach enough context for them to understand the work, such as a description and the checklist to use.

The team checklist should be one page, like your personal checklist. However, the team checklist should be filled with issue types that concern your team—for example, particular security, reliability, logic, or failure handling issues that your team's software is most prone to have. Like your personal checklist, the team checklist should be regularly updated to remove issues that are no longer prevalent (the team has improved) and to add arising issues.

The inspectors then carefully and independently inspect the completed work for the issue types on the checklist, noting the type and line number for each issue. Often inspectors will find issues that aren't on the checklist. Those issues should be noted as well. Issue types that are prominent but missing from the checklist should be researched for the best way to eliminate them in the future (possibly by updating the unit testing, automated code analysis, or inspection checklist).

There are many approaches to inspecting. One effective technique is to look through the entire work for each issue type, one at a time, then move to the next issue in the checklist. Sure, it sounds monotonous, but the lack of context switching makes it an easy and effective method. Regardless of their approach, after a little practice, inspectors get very good at catching a huge percentage of the issues.

By using the same checklists, the inspectors are ready to merge issue lists and determine how many issues they found in common.

Magical merge meeting

At the inspection meeting, the inspector who found the most issues fills out a spreadsheet, listing all his issues. Often an appointed moderator who is experienced with the process handles the data entry and keeps the meeting on track.

Online Materials Inspection worksheet (InspectionWorksheetExample.xls; InspectionWorksheetTemplate.xlt)

For each issue found, the group of inspectors note who else found the same issue (type and occurrence). Multiple issues can be found on the same source line. In addition to agreeing on duplicates, the inspectors also mark whether or not the issue must be fixed or if it should be left to the discretion of the author.

When the first inspector finishes his list, the next inspector follows the same procedure, skipping over issues already noted by the first inspector. This continues until all issues are captured in the spreadsheet. The spreadsheet then automatically computes statistics like defect density, yield, the total number of issues found, and the likely number of issues still lurking in the work.

How is that possible? Because everyone looked at the same work for the same issues and you captured which issues were found and missed by each person. So you can accurately estimate how many total issues were missed overall. You are no longer FOCKED.

Eric Aside The math is based on the old capture-recapture technique used to estimate the number of fish in a lake. You catch *n* fish at various lake locations, tag them, and release them into the lake. Later you catch another bunch of fish at the same locations and note the percentage tagged. Then you divide *n* by that percentage and you have a good estimate for the total number of fish. For inspections, the first inspector's issues are considered "tagged." The other inspectors "fish" for issues in the same design or code "lake." The results give a simple yet accurate estimate of the total number of issues.

Tricks of the trade

Naturally, there are lots of little tricks to help you get the most out of inspections:

- While you can do inspections with large groups, typically you only need two or three inspectors to get great results (not counting the author). Start with three to five inspectors to get the hang of it, and then drop the number as people gain skill.

- Just like informal reviews, it takes time to do inspections right. Plan three to five pages per hour for docs and 200–400 lines per hour for code. These are reasonable rates. Faster and you miss issues, slower and your head turns to mush.
- Don't inspect too much at once. A thousand lines of pure new code will take around three or four hours to inspect. Be kind and inspect it in chunks.
- Do not discuss fixes at the meeting. Inspections are for issue detection, not for generating ideas. Don't let yourselves get FOCKED.
- It helps to have a moderator experienced at filling out the spreadsheet and keeping people from discussing fixes.
- The moderator can also be used to verify that work is complete and ready for inspection. Until people get the hang of it (including people new to your group), you should definitely have a moderator.
- Your team should have a quality bar for how many issues are acceptable to pass on for further testing. If the results of your inspection indicate the number of must-fixes remaining is too high, the authors must have their work re-inspected after fixing the issues found. A nice guideline is: any yield under 75% requires re-inspection.

Getting it right

When used properly, inspections can decrease by orders of magnitude the number of issues released to customers. Teams, both inside and outside Microsoft, have seen these results repeatedly. The key is to not confuse your goals for orchestrating collective knowledge.

If you want to generate some ideas or solutions, run a brainstorming meeting, keeping an open mind and avoiding criticism. If you want early feedback, ask for an informal review, allowing people some flexibility and expecting that some issues will be missed. If you want to assess quality and detect issues, organize an inspection, focusing on the findings and not the fixes. Remember that you can always use brainstorming or informal reviews to help you fix the issues you find.

A little communication and clarity can go a long way. Knowing when to apply which techniques for group engagement can take the failure out of FOCKED, and we all know what a profitable strategy that can be.

October 1, 2006: "Bold predictions of quality"

I've been busy dogfooding lately. It's an ideal diversion for masochists. When it gets to be too much, I can always take respite in a nice horror film. Thank goodness what passes for dogfood now is a vast improvement over years past.

Eric Aside *Dogfooding* is the practice of using prerelease builds of products for your day-to-day work. It encourages teams to make products right from the start, and it provides early feedback on the products' value and usability.

Years ago, running a dogfood build and having your machine unplugged were almost indistinguishable in terms of productivity. These days, substantial parts of dogfood builds are fully functional, while others remain unusable, unreliable, or unconscionable. This begs the question, "Why?"

Why are some portions of new builds solid, thoughtful, and engaging, while others remain flaky, unfathomable, and exasperating? How can that be? Ask managers and they'll say, "Well, it's tough to tell ahead of time what's going to be good or rancid." Sounds like they're washing down my dogfood with male bovine dung.

Enigma? I don't think so

Software quality is unpredictable? Don't make me gag. Poor-quality software has all the subtlety of a neighborhood ice cream truck. You know it's bad for you; you know it's coming a mile away; yet you can't resist. Managers choose to ignore the signs and buy the ice cream (poor software) because they hate disappointing the children (upper management) and can't resist the instant gratification ("progress").

We've gotten so used to poor software that many people have forgotten the early signs. Let me summarize the rest of this column and make it simple for you. Good software is solid and originated out of complete and critical customer scenarios. Bad software is buggy and originated out of someone's behind.

Twins of evil

How do you spot bad software before it's integrated into the main branch? First, remember there are two aspects to quality—engineering and value. Most engineers get caught up in the engineering side of quality—the bugs. However, flawlessly engineered features can be glorified crud to customers if the ideas came from the wrong place. I talk about this more in "The other side of quality—Designers and architects" in Chapter 6.

We're looking to predict both buggy code and code with questionable pedigree. Predicting buggy code is easier, so we'll start there.

The usual suspects

In 2003, Pat Wickline studied the root causes of late cycle bugs in Windows Server 2003. The results were similar to his 2001 study of bugs in Windows 2000 Hotfixes. Simply put, more than 90% of bugs could have been found by design reviews, code reviews, code analysis (like PREfast), and planned testing. No one method would have found every bug, but the combination nearly finds them all.

In 2004, Nachiappan Nagappan studied measurable attributes in an engineering system that correlated well to bugs found later. Those attributes were code churn (the percentage of code lines added or changed per file) and code analysis results (the number of PREfast or PREfix defects found per line of code).

Eric Aside He's updated his thinking to focus on churn, complexity measures, and organizational structure. In 2008, Nachiappan published a finding that the attribute with the highest correlation to defects was organizational structure. Organizations aligned with the component architecture had the fewest defects (components tended to be edited by single teams). Organizations aligned in ways that resulted in multiple teams editing the same components had the most defects. I believe this research finding is among the most important in software management history, but I don't want to understate it.

If you want to prevent poorly engineered code from getting into the main branch, have your build track code churn and code analysis results. If those measures go beyond the norms for quality code, then reject the check-in. If your developers don't like it, tell them to do more design and code reviews and write more unit tests.

"What if there's too much code churn, but the feature enables a complete and critical customer scenario?" you might ask. Allow me first to congratulate you on coming up with the only decent reason to not junk the code entirely. Then junk the code entirely. It's time for a rewrite of that section. That's the only way the feature will ever reach your engineering quality goals.

You're gonna love it

Let's move on to predicting questionable feature pedigree. Buggy code is easy to measure and control, though it does require management to set a bar and stick to it. The value of software is harder to measure, but in the end it requires the same thing—management must set a bar and stick to it.

How do you know if a feature or check-in will really be valued by customers? That's easy. If it's part of a complete and critical customer scenario, then users will love it. How do you know if a scenario is complete and critical? That's the hard part.

Luckily, you don't have to do that work. We pay marketing, product planning, and upper management to figure out the complete and critical customer scenarios for a release. No one feature team or product group can do it, because complete scenarios cut across product groups. Instead, engineering's job is to tell the planners what's possible, and then solidly implement the planned critical scenarios from end to end.

Quit fooling around

Of course, overzealous engineers of all kinds, not just PMs, will try to sneak in features that aren't part of planned complete and critical scenarios. While doing so might relieve that engineer's creative constipation, what comes out is predictably putrid for customers.

To trap poorly conceived features before scarce resources get wasted, you must take two steps:

1. Have a clearly documented vision or value proposition that lists the complete and critical scenarios for the release. Prototypes, personas, user experience designs, and high-level architectures also help clarify what's needed immensely.
2. Convene a governing board who owns the vision or value proposition and have them review every feature. If the feature doesn't fit a complete and critical scenario, it's cut. Period. At the beginning of each major milestone, every GPM reviews their list of upcoming features with the governing board. While the board may not review every feature in great detail, they must still ruthlessly and relentlessly uphold the quality, value, and integrity of the release.

Eric Aside The best groups at Microsoft have been following this process for years.

These two steps precisely correspond to setting the bar and sticking to it. While this bar is more subjective than the engineering quality bar, both require the same disciplined commitment by management to be successful.

Quality is no accident

It's not difficult to predict quality. In fact, it is straightforward. Yet managers at all levels rarely apply the rigor necessary to assure quality.

Maybe managers are afraid assuring quality will add too much time to the schedule. As if doing it right the first time and sticking to only the critical needs takes longer. In fact, when quality is what customers expect, then focusing on quality is always the fastest way to ship.

Maybe managers are afraid engineers won't like assuring quality. As if engineers take no pride in their work or enjoy ambiguity and wasting their time. In fact, engineers take great pride in the quality of their work, prefer to know what's expected, and hate wasting effort.

The truth is that quality is expected, quality is fundamental, quality is central to our success. It is because our customers say it is.

Quality is the right thing to do and the right way to do it. It is the key to our future survival and prosperity. Quality is no accident. You can predict and control it. All you need is a brain and a backbone. Get yours today.

Eric Aside While they both vigorously advocate quality, it's worth noting the differences between "Where's the beef? Why we need quality" and "Bold predictions of quality." The first discusses why quality is needed and the mechanics of getting it. The second describes how to measure and refine your work to push the quality bar higher. We've made significant progress, but quality is an ideal that demands eternal vigilance.

May 1, 2008: "Crash dummies: Resilience"

I heard a remark the other day that seemed stupid on the surface, but when I really thought about it I realized it was completely idiotic and irresponsible. The remark was that it's better to crash and let Watson report the error than it is to catch the exception and try to correct it.

Eric Aside A lot of people flipped out that I thought you should catch the exception. The more thoughtful readers pointed out security concerns with handling exceptions and the dangers of continuing an application with corrupted state. I couldn't agree more. If the failure or exception leaves the program compromised you can't simply continue. My point is just failing and giving up is wrong for users. One solution I talk about below is to fail, report, and reboot (restart) the application, like Office now does.

Watson is the internal name for the functionality behind the Send Error Report dialog box you see when an application running on Windows crashes. (Always send the error; we truly pay attention.)

From a technical perspective, there is some sense to the strategy of allowing the crash to complete and get reported. It's like the logic behind asserts—the moment you realize you are in a bad state, capture that state and abort. That way, when you are debugging later you'll be as close as possible to the cause of the problem. If you don't abort immediately, it's often impossible to reconstruct the state and identify what went wrong. That's why asserts are good, right? So, crashing is sensible, right?

Eric Aside An *assert* is a programming construct that checks if a relationship the programmer believes should be true is actually true. If it isn't true, assert implementations typically abort the program when debugging and log an error when running in production. Asserts are commonly used to check that parameters to a function are properly formed and to check that object states are consistent.

Oh please. Asserts and crashing are so 1990s. If you're still thinking that way, you need to shut off your Walkman and join the twenty-first century, unless you write software just for yourself and your old-school buddies. These days, software isn't expected to run only until its programmer got tired. It's expected to run and keep running. Period.

Struggle against reality

Hold on, an old-school developer, I'll call him Axl Rose, wants to inject "reality" into the discussion. "Look," says Axl, "you can't just wish bad machine states away, and you can't fix every bug no matter how late you party." You're right, Axl. While we need to design, test, and code our products and services to be as error free as possible, there will always be bugs. What we in the new century have realized is that for many issues it's not the bugs that are the problem—it's how we respond to those bugs that matters.

Axl Rose responds to bugs by capturing data about them in hopes of identifying the cause. Enlightened engineers respond to bugs by expecting them, logging them, and making their software resilient to failure. Sure, we still want to fix the bugs we log because failures are costly to performance and impact the customer experience. However, cars, TVs, and networking fail all the time. They are just designed to be resilient to those failures so that crashes are rare.

Eric Aside This was among my most controversial columns. Three years later, it's clear the readers fell into two rough categories:

- Readers who couldn't get past the specific technical details about nonabortive asserts and restarting processes after unhandled exceptions. They hated the column and generally questioned my mental state.
- Readers who understood this column wasn't about asserts and exceptions—it was about resilience. They were challenged by the column to improve the user experience of our products and services (some even liked it).

Perhaps be less assertive

“But asserts are still good, right? Everyone says so,” says Axl. No. Asserts as they are implemented today are evil. They are evil. I mean it, evil. They cause programs to be fragile instead of resilient. They perpetuate the mindset that you respond to failure by giving up instead of rolling back and starting over.

We need to change how asserts act. Instead of aborting, asserts should log problems and then trigger a recovery. I repeat—keep the asserts, but change how they act. You still want asserts to detect failures early. What’s even more important is how you respond to those failures, including the ones that slip through.

Eric Aside Just once more for emphasis—using asserts to detect problems early is good. Using asserts to avoid having to code against failures is bad.

If at first you don’t succeed

So, how do you respond appropriately to failure? Well, how do you? I mean, in real life, how do you respond to failure? Do you give up and walk away? I doubt you made it through the Microsoft interview process if that was your attitude.

When you experience failure, you start over and try again. Ideally, you take notes about what went wrong and analyze them to improve, but usually that comes later. In the moment, you simply dust yourself off and give it another go.

For web services, the approach is called the five Rs—retry, restart, reboot, reimage, and replace. Let’s break them down:

- **Retry** First off, you try the failed action again. Often something just goofed the first time and will work the second time.
- **Restart** If retrying doesn’t work, restarting often does. For services, this often means rolling back and restarting a transaction or unloading a DLL, reloading it, and performing the action again the way Internet Information Server (IIS) does.
- **Reboot** If restarting doesn’t work, do what a user would do, and reboot the machine.
- **Reimage** If rebooting doesn’t work, do what support would do, and reimage the application or entire box.
- **Replace** If reimaging doesn’t do the trick, it’s time to get a new device.

Welcome to the jungle

Much of our software doesn’t run as a service in a datacenter, and contrary to what Google might have you believe, customers don’t want all software to depend on a service. For client software, the five Rs might seem irrelevant to you. Ah, to be so naïve and dismissive.

The five Rs apply just as well to client and application software on a PC or a phone. The key most engineers miss is defining the *action*, the scope of what gets retried or restarted.

On the web it’s easier to identify—the action is usually a transaction to a database or a GET or POST to a page. For client and application software, you need to think more about what action the user or subsystem is attempting.

Well-designed software will have custom error handling at the end of each action, just like I talked about in my column “A tragedy of error handling” (which appears in Chapter 6). Having custom error handling after actions makes applying the five Rs much simpler.

Unfortunately, lots of throwback engineers, like Axl Rose, use a Routine for Error Central Handling (RECH) instead, as I described in the same column. If your code looks like Axl’s, you’ve got some work to do to separate out the actions, but it’s worth it if a few actions harbor most crashes and you aren’t able to fix the root cause.

Just like starting over

Let’s check out some examples of applying the five Rs to client and application software:

- **Retry** PCs and devices are a bit more predictable than web services, so failed operations will likely fail again. However, retrying works for issues that fail sporadically, like network connectivity or data contention. So, when saving a file, rather than blocking for what seems like an eternity and then failing, try blocking for a short timeout and then trying again—a better result for the same time or less. Doing so asynchronously unblocks the user entirely and is even better, but it might be tricky.
- **Restart** What can you restart at the client level? How about device drivers, database connections, OLE objects, DLL loads, network connections, worker threads, dialogs, services, and resource handles. Of course, blindly restarting the components you depend upon is silly. You have to consider the kind of failure, and you need to restart the full action to ensure that you don’t confuse state. Yes, it’s not trivial. What kills me is that as a sophisticated user, restarting components is exactly what I do to fix half the problems I encounter. Why can’t the code do the same? Why is the code so inept? Wait for it, the answer will come to you.
- **Reboot** If restarting components doesn’t work or isn’t possible because of a serious failure, you need to restart the client or application itself—a reboot. Most of the Office applications do this automatically now. They even recover most of their state as a

bonus. There are some phone and game applications that purposely freeze the screen and reboot the application or device in order to recover (works only for fast reboots).

- **Reimage** If rebooting the application doesn't work, what does product support tell you to do? Reinstall the software. Yes, this is an extreme measure, but these days installs and repairs are entirely programmable for most applications, often at a component level. You'll likely need to involve the user and might even have to check online for a fix. But if you're expecting the user to do it, then you should do it.
- **Replace** This is where we lose. If our software fails to correct the problem, the customer has few choices left. These days, with competitors aching to steal away our business, let's hope we've tried all the other options first.

Eric Aside Though I got plenty of unflattering mail for this column, many of the suggestions in this section were implemented in Windows 7, making that release far more resilient. I can't take any credit for that—my point is that these notions can be applied effectively and really improve our customers' experience. We've still got ample opportunities to get better in all our products.

Let's not be hasty

Mr. Rose has another question: "Wait, we can't just unilaterally take these actions. Customers must be alerted and give permission, right?" Well Axl, that depends.

Certainly, there are cases where the customer must provide increased privileges to restart certain subsystems or repair installs. There are also cases when an action could be time-consuming or have undesirable side effects. However, most actions are clear, quick, and solve the problem without requiring user intervention. Regardless, the key word here is *action*.

There's no point in alerting the user about anything unless it's actionable. That goes for all messages. What's the point of telling me an operation failed if there's no action I can take to fix it or prevent it from happening again? Why not just tell me to put an axe through the screen? If there is a constructive action I can take, why doesn't the code just take it? And we have the audacity at times to think the customer is dumb? Unbelievable.

It's always the same

"Fine, this is extra work though," complains Axl, "and who says the software won't just be retrying, restarting, rebooting, and reimaging all the time? After all, if the bug happened once, it will happen again." Actually Axl, bugs come in two flavors—repeatable and random. Some people call these Bohrbugs and Heisenbugs, respectively.

Eric Aside The terms Bohrbug and Heisenbug date back before the 1990s. Jim Gray talked about them in a 1985 paper, "Why Do Computers Stop and What Can Be Done About It?"

Using the five Rs will resolve random bugs, rendering them almost harmless. However, repeatable bugs will repeat, which is why logging these issues is so important. Even if the program or service doesn't crash, we still want the failure reported so we can recognize and repair the repeatable bugs, and perhaps even pin down the random bugs. The good news is that the nastiest bugs in this model, the repeatable ones, are by far the easiest to fix.

By putting in some extra work, we can make our software resilient to failure even if it isn't bug free. It will just appear to be bug free and continue operating that way indefinitely. All it takes is a change in how we think about errors—expecting, logging, and handling them instead of catching them. Isn't that worth the kudos (and free time) you'll get from family and friends when our software just works? Welcome to the new world.

Eric Aside I don't expect this new approach to happen tomorrow. It's a big change, particularly in the client and application areas. It used to be that only geeks had computers, so users knew how to restart and repair drivers. Now, everything just has to work with little or no user intervention. Part of the solution is higher engineering quality, but that only goes so far. There will always be failures even if the code is bug free. Resilience to failure is the clear next step.

October 1, 2008: "Nailing the nominals"

People are always looking for that amazing breakthrough technology or process that solves all their problems—enhances their love life, trims their waist, and improves the productivity of their development team. That's why process manias like Agile and Six Sigma are so enticing. Just splat the *Scrum* tag on your development team and "bam!"—your team is suddenly ten times as prolific.

It would be hilarious if it weren't true in the minds of the narcissistic ninnies with an addiction to magic methodologies. I'm happy that people are willing to try something new. I just wish they wouldn't confuse new with necessary, or minor fixes with major problems.

The truth is that there's nothing magic about producing great software with high quality on time. (And no, software services aren't different in this regard.) Human beings have had large teams building complex engineering marvels for thousands of years. There is no magic. There is solid design and disciplined execution. Period.

Eric Aside When your code base is less than 100,000 lines, and you've got fewer than 15 people on the project, you don't need solid design and disciplined execution. You can wing it—use emergent design, have a loose upfront design bar, rewrite and refactor the code endlessly while the customer looks over your shoulder. When your code base and your project are bigger, it's solid design and disciplined execution or it's broken code, broken teams, and broken schedules.

Back to basics

I worked at Boeing for five years before coming to Microsoft. The areospace industry uses the terms *nominals* and *tolerances* to refer to acceptable values and tight refinements. A nominal might be a support beam being placed two feet from a heating duct. A tolerance would be that location give or take 1.5 inches.

There were always inexperienced engineers who concerned themselves with tolerances. They were constantly looking for a clever technique to give them an extra fraction of an inch of tolerance, as if that mattered. Real engineers who designed and built the airplanes knew these tolerance tinkerers were misguided, out-of-touch, naïve fools who missed the point entirely.

The key to building an airplane isn't tinkering the tolerances—it's nailing the nominals. It's not, "Does the steel stringer bend within 0.058 inches of the specified skin?" It's, "Does the steel stringer pass right through a cooling vent?" Stop worrying about the fancy details and get the basics right, ding-a-ling!

Eric Aside You might think I'm criticizing Boeing, but it's quite the opposite. When lives are in the balance, it's important to get the basics right—getting the passengers to their destination safely. If the stringer is a few millimeters off, you can shim it. If two key systems run through the same space, a minor accident can lead to a major catastrophe. Never let the details distract you from what customers truly care about.

I want to be a cowboy

The software industry suffers greatly from dingbats tinkering tolerances instead of nailing nominals. You can see it in most software, regardless of how or where it ships. The focus of the engineering was clearly on all kinds of fancy little features instead of getting the software to work right for basic cases.

Maybe the basic cases don't appeal to engineers. Maybe engineers can relate only to tinkerers, not to folks who want solutions that are simple and reliable. However, I don't think so.

I think nailing the nominals requires solid design and disciplined execution, both of which require software engineers to put the project and the customer ahead of their personal interests. Unfortunately, that's not what cowboy coders crave, which is why cowboy coders should either grow up or move out.

But it's so simple

Still here? Still believe the customer and project should come first? Still realize that true customer value only comes when you act as a team, putting the customer at the center instead of yourselves? Good, let's talk about the nominals of software.

As I said, the nominals revolve around solid design and disciplined execution. But that's true for any significant engineering venture. What does it mean for software? I'll keep it simple.

Solid design means:

- Understanding what the customer is trying to accomplish with your software (product planning, value proposition/vision).
- Thinking through the end-to-end experience, including pitfalls (experience design, scenarios).
- Decomposing that experience into distinct engineerable pieces (engineering design, architecture).

Disciplined execution means:

- Ordering work based on priority and dependency.
- Creating a schedule based on data from past projects.
- Establishing and upholding completion criteria for every step.

In the simplest of terms, the nominals are "**think before you act**" and "**define done**."

Eric Aside Simple right? Think ahead and define done. Who wouldn't? How about most of the projects I see inexperienced teams attempt. Even experienced teams rush to implementation without clear scenarios, architecture, and completion criteria when some idiotic executive gets a bee up his bum.

Is it done?

I've written about experience and engineering design many times before (particularly in "The other side of quality—Designers and architects" in Chapter 6), so this time let's drill down on defining *done*. For a distinct engineerable piece of the architecture, be that a feature, component, API, or web service, what should the completion criteria be?

You could set bug limits or test code coverage, but those evaluate intermediate results. What do you care about at the end? What should the final result of an engineerable piece of the architecture be? Well, it should fully satisfy its part of the end-to-end scenario, including performance targets and avoiding pitfalls, and it should be secure and reliable.

What causes software to fail scenarios and reliablity targets? Smart dedicated people, at Microsoft and around the globe, have been studying this problem for decades. The net result, which you can read in every study ever done, is that problems stem from failing to complete only five practices—five practices that should define done:

- **Documented design.** Think before you act and capture it so that peers can do a…
- **Design review,** which catches up to half the errors before you write a line of code, which you then…
- **Code review,** which, if you use inspections, can catch more than 70% of errors, though some errors will only be caught by…
- **Code analysis,** like lint or PREfast, which catches patterns humans can miss but doesn't entirely replace…
- **Unit and component testing,** which catch the remaining issues, including stress, performance, and edge cases, as well as end-to-end scenario checks.

Without defining *done*, tasks never finish (sound familiar?). Even worse, cowboy coders can claim to be finished with no accountability for quality work. Ever wonder why great developers end up cleaning up after poor developers? It's because poor developers claim completion for crud. Without defining done, who else can keep their crappy code from customers?

Eric Aside I left out all the different forms of system test because I'm defining done for a component not the overall product. Clearly, there are many other steps before a product or service is ready for release.

It's not that complicated

Notice how none of the five practices that define done are terribly exotic or revolutionary? They are basic. Everyone knows you should do them. Yet engineers will fall over themselves to become certified Scrum Masters, create cool new tools, or learn the latest technology before they'll actually run design reviews or write comprehensive unit and component tests.

Nail the nominals before tinkering the tolerances. It's not magic. It's not complicated. It's not hard. It takes discipline. It takes putting the customer and the project before yourself. It takes being a professional instead of a chump. Think you can handle it?

Chapter 6

Software Design If We Have Time

In this chapter:

One of the basic questions an engineering leader has to answer is, "What software development improvement would have the largest impact on the quality and value of our products?" My initial response would be broad and consistent measurement of desired outcomes, because they tell you the right improvements to make and the impact of making them. It takes out the guesswork and provides positive team-based motivation.

However, if I had to speculate what the measured outcomes would tell us, I'd guess feature crews (Lean development) and better design would have the biggest impacts. (We're already doing reviews, code analysis, and unit testing.) I spent the second chapter talking about Lean development. It's time to talk about better design.

In this chapter, I. M. Wright exposes the fundamentals and intricacies of software design. The first column starts with basic error handling. The second one denounces code and data replication. The third column outlines the complete design process, including suggested practices. The fourth column focuses on designing great user experiences and making them real. The fifth column reveals the true value and purpose of architecture, beyond keeping the architect occupied. The sixth column looks at software performance design from the user's perspective. The seventh column covers designing real value with services. The eighth column examines the world of prototyping, and the last column appeals to self-restraint in green fields.

Before getting into the columns, I'd like to address the two biggest excuses software engineers give for avoiding design work: the design will emerge, and there isn't *enough time. Emergent design claims up-front design is a waste; instead, you should discover and refactor the design as you go. That's great for small code bases (less than 100,000 lines), when refactoring isn't a big deal. However, most production code bases are far larger, and serious rework is extraordinarily expensive. Not thinking things through before you refactor or make other changes can cost you dearly. So take the time now or you're sure not to have enough time later.*

—Eric

September 1, 2001: "A tragedy of error handling"

If there is any single aspect of our production code that has been traditionally and uniformly lame, it's error handling. Office made some great strides in this area for Office 2000 with its LAME registry setting for error dialogs. Windows 2000 also improved things quite a bit by working hard at providing meaningful messages with actionable directions. Both Office XP and Windows XP now automatically report severe errors back to us to evaluate and track. However, these efforts only bring us from kicking our users when they're down to apologizing for slamming them in the first place and then perhaps handing them a cane to push themselves upright.

Eric Aside The internal Office 2000 LAME registry setting added a "lame" button to every Office 2000 error dialog. If you didn't like the dialog, you could hit the "lame" button and your "vote" would be recorded. These days it's been replaced by the "Was this information helpful" link all users can access at the bottom of Office error dialogs.

So why doesn't the code fix the error itself? From my years of staring at our code, I see two main reasons. First, the code has no idea what went wrong in the first place. Second, the error-handling code is not in a position to fix the problem, even if it knew what it was. These two problems are related and pervasive. Let's talk a bit about the typical situation.

The horror, the horror

A dev writes a bunch of code. Another dev adds a bunch more. Then they add code from some other group. Then they add more. Then they realize that they need to handle errors, but they don't want to go back and put error handling everywhere, so they write a Routine for Error Central Handling, or RECH, and propagate errors to it. Then they write the next version that adds more code, maybe written by completely different people. Some folks return meaningful errors, some return a simple pass/fail. Some folks like exceptions; some folks like error codes. The RECH only works with exceptions or error codes, not both.

Eric Aside I made up the RECH acronym for this column. I have no idea what the actual name is, if any, for the unfortunate practice it references.

If the RECH works with exceptions, then sections of code that return error codes are wrapped with a throw on failure. If the RECH works with error codes, then sections of code that throw exceptions are wrapped with a generic catch that returns an error code. If a section of code returns pass/fail, then it gets wrapped with a generic exception or error code, which might be converted later to an error code or exception. Even if you start with a function that returns descriptive error codes, like much of Win32 or OLE, these calls are often wrapped by a function that either reduces the codes to pass/fail or ignores them completely. All along the way, information is lost…forever.

Taking exception

To add pain to agony, the mixing of exceptions with error codes is a disaster. You can't use exceptions if you don't unwind your stack objects properly. This usually means that anything requiring non-trivial destruction must be an object. That's easy in C#, but it's hard in C and C++.

For example, you must use smart pointers exclusively in code that throws exceptions. However, if some parts of the code use exceptions and others don't, you often get a situation in which an exception-throwing function is called from an error-code-returning function and the catch doesn't happen till you're three levels up. Thus an error generates more errors and corrective action is compromised. It gets even worse in multithreaded apps where exceptions must be handled per thread.

Don't lose it, use it!

You want to use the error information you have in the best possible way. So what do you do? First, pick your poison and go with it.

- If you want to use exceptions, fine, but use them everywhere and make everything non-trivial an object (the .NET Framework model).
- If you want to use error codes, fine, but propagate them in a lossless fashion or handle them at the source.
- If you really must mix metaphors, wrap every exception call inside a return code function with your exception-handling routine, and wrap every return code function call

inside an exception function with your return-code-handling routine. Now at least you are not losing data.

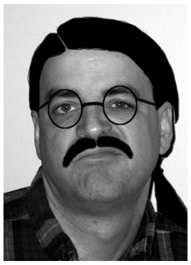
Eric Aside Please note, this doesn't mean you should wrap all exceptions for the sake of wrapping them. That's inefficient and quite strange. But say you are going to hide an exception-raising call within an outer function that isn't supposed to raise exceptions. To uphold the outer function's contract, you need to catch exceptions from the inner function and convert them into an appropriate error code. Ideally, you aren't using a mixed model in the first place.

Next, you need to plan for action when an error occurs. Figure out the highest level at which you still know what to do if a section of code fails. That level is rarely the top, so stop RECHing.

At that highest actionable level, add your error handling. This will add more than just one error-handling function to your code, but it probably won't add a thousand. The key is to go as high as you can in the stack, but no higher. When using error codes, you may need to add buffers to your application object to hold key information like file paths or flags so that your error handling can fix the problem.

Only as a last resort—or as an act of bravado—report the error to the user. The net result is a system that always seems to work or at least always seems to care. Our customers will love us for it.

February 1, 2002: "Too many cooks spoil the broth—Sole authority"



Segall's Law says, "A man with one watch knows what time it is; a man with two watches is never quite sure." While most people intuitively understand this for watches, it's apparently unfathomable to most devs when it comes to programming. Our code is littered with repetitions of algorithms and data, not only across the company, but also within single applications.

It seems that everyone wants to craft their own watch. We store the same user or system data many different ways with no hope of reconciliation. We copy and paste code all over the place, and then we let it get out of sync. The whole idea of different teams sharing code is ludicrous if we can't even manage to share code within the same application.

A picture is worth a thousand words

Don't believe me? Open Windows Explorer and find a directory with a .jpg or .gif file. Open a blank slide in PowerPoint. Copy the .jpg or .gif file and paste it to the slide background. Now paste it to the text area. So far, so good.

Now drag the .jpg or .gif file and drop it in the background. Next drag and drop it in the text area. Whoops! Try this same experiment with formatted text. Notice that you get an action on paste, but no action on drop.

Eric Aside It looks like PowerPoint fixed the image bug in Office 2007, but the problem with formatted text remains.

I'm not trying to single out PowerPoint. Try it with other applications and other data. You'll notice how data from a paste is treated differently than data from a drop, and it's not just due to where it's placed. Each of these scenarios should act the same. The data is identical in all cases. In fact, paste and drop do act the same in Word (nice job).

So why does the same data inserted into a document in roughly the same way produce different results? Well, duh! It's a different code path. The irony here is that duplicate code paths are often created by copying and pasting the original code itself.

Does anyone really know what time it is?

Having two code paths to do the same work is just like having two watches. Maybe they behave the same, maybe they don't. Keeping the code paths synchronized with bug fixes and spec changes is a nightmare.

Of course, the solution is to write one routine that does the work and then call that routine from each code path. That's first-year computer programming fundamentals, and yet we make this freshman mistake over and over again.

But the problem doesn't begin and end with functions. It extends to data as well. How many bugs have you fixed that were caused by out-of-sync data values? Could these data values have been derived from one value, but were instead kept separately? And don't even get me started on how often I'm forced to type my e-mail address every day.

Maintaining two watches and keeping them both accurate requires manual synchronization. This is a tedious and error-prone process. It's much easier to have a single watch. Unfortunately, many of us have a watch, a PC, and a Pocket PC, plus we work with a whole mess of people who have their own timepieces.

Luckily, your PC can synchronize automatically with a network synced to an atomic clock. Then your Pocket PC, your watch, and everyone else's can be synced to your PC's time. Automate all that and the problem is solved.

There can be only one

There is a fundamental principle at work here. I call it "sole authority." Every piece of data should have a sole authority for its value. Every operation should have a sole authority for how it is done. Keeping all your timepieces in sync is possible because there is a sole authority for the current time. Consistent user interface is possible if there is a sole authority for how each operation is completed—for things like painting, data input, and message handling.

Every time you copy and paste code, stop and think, "Am I repeating myself or is this really a different operation? What is the sole authority for how this operation is conducted?" Then act accordingly.

When you are about to create yet another registry entry, stop and think, "Is this really a whole new concept to the user or system, or can I derive this value from another authority?"

If you are asked to create a custom control, think, "Is there an existing authority on how this control should work, and how can I use it?"

For you performance geeks out there, remember that caching a computed result is often slower than recomputing the result, due to faster processing speeds and the gating effect of cache hits.

Everything is connected to everything else

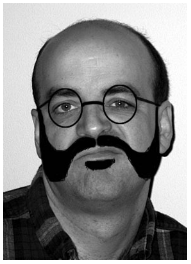
Sole authority becomes much more important in the .NET world. We are hard at work giving our users one authority for their identity, another authority for their calendar, one for their address book, and so on. We are betting that the resulting convenience will be a boon to customers and a boost to our bottom line.

But it only works if we use one method to perform all the associated operations. Doing otherwise not only opens us to bugs and makes our users confused and frustrated, but it also can multiply security holes and increase operational costs.

Eric Aside While the idea of having one authority for identity, calendaring, and so on was good in concept, concerns around trust, privacy, and interoperability caused the notion to fall short in the marketplace.

In software development, too many cooks do spoil the broth. Spoiled broth can give our customers and partners dysentery, not to mention what it does to the cooks. By simply questioning what code or data is the sole authority for every operation or query, we can simplify our code, make it easier to fix and maintain, and give our customers and partners a consistent, pleasant, and intelligent experience.

May 1, 2004: "Resolved by design"



There you are, sitting in front of a whiteboard while some obsessive dweeb goes on incessantly about some design issue. You try to ground the discussion and bring the conversation back to the actual work, but this neurotic nag won't let anything go. "What if this happens?" "We haven't considered this." "We don't know that." "We can't start yet." Analysis paralysis. Meanwhile the clock is ticking and you've got to ship.

We all know design is important. We all know quality design is essential to quality products. But darn it, at some point you've got to code. At some point you've got to ship. The best design in the world is just wallpaper if you can't get it coded in time, and I don't hear many managers saying, "Take all the design time you need."

What is good enough?

Yet the other extreme is just as evil. Jumping in and coding with just a token of forethought on a fleeting whiteboard leads to tremendous rework, tons of bugs, and a fragile framework for future development. Most analysis of defects reveals between 40–50% of them were in the design. Skipping the design step is no solution, but how much analysis is enough? You know when you're done coding—when is the design sufficient to begin development?

To me, the lack of a well-defined design process is the single most debilitating developer debacle. It's not like you don't know it's important—coding is far easier, faster, and dramatically less error prone with a clear and complete design. No worries, no quandaries, no surprises, and none of the infamous Homer Simpson, "Doh!" But did you learn a design process in college? No, they don't teach it. Did you learn one when joining Microsoft? Not likely; most groups are pretty ad hoc about design, and many hardly practice design at all. Ah, but your good buddy I. M. is here to help.

Eric Aside I lay out a series of design steps and approaches below that cover all design aspects. However, how do you know which steps you can skip, and how do you know when you've completed a step? After all, you want to do just enough design—not too much or too little. The answer is confidence—you want to proceed to implementation with confidence. For each step you ask, "Are we confident about this design aspect?" If so, you can skip it. If not, you work on that aspect until you are confident. Be honest with yourself and your team. You don't need to understand every detail, but you do need to know what you're doing.

Design complete

The most important aspect of a design process is completeness. Completeness tells you what is enough. You don't want to do too much or too little. Software has multiple dimensions. There are internal workings and external interfaces. There are static concepts and dynamic interactions. All of these dimensions must be covered to create a complete design. Additionally, there are different design tools to help you work with the different areas and levels of abstraction. Here's a table to help make sense of this.

Dimensions of software design

	External	Internal
Static	PM spec	Dev spec
	API definition	Test-Driven Development
Dynamic	Use cases	Sequence diagrams
	Scenarios and personas	State diagrams, flow charts, threat and failure modeling

Watts S. Humphrey showed me this table while he was visiting Microsoft once. My team and I filled it in with common design practices and the movement from quadrant to quadrant.

Most design processes spiral inward with a clockwise flow from the lower-left quadrant of the table, starting with the high-level design steps:

- **Scenarios and personas** Describes at a high level how customers will interact with the software.
- **PM spec** Describes the external pieces of the puzzle, which include requirements, dialog boxes, menus, screens, data collection, and other features.
- **Dev spec** Specifies the class hierarchy and relationships, component stack and relationships, and anything else needed at a high level to describe the structure of the implementation. This spec is also referred to as the design document (a misnomer, given its limited scope) or architecture document.
- **State diagrams, flow charts, threat and failure modeling** Describe and control complex interactions between the objects and components in the system.

A few comments on these initial, high-level design steps:

- Each step has a limited, well-defined scope. By separating and covering all four design quadrants, you won't need to rely on lots of words and repetition to fully describe the design. Be brief, reuse prior examples, and rely on simple diagrams or pictures where appropriate.
- Not every step is necessary in every design. Some designs have little external interface. Some designs don't carry state. Use your own best judgment.

- The external quadrants have a wide audience (all disciplines). The internal quadrants have a purely technical audience.
- Unified Modeling Language (UML) provides ready-to-use diagrams for many of these areas, and there are tools that make creating UML diagrams easy. For example, Visio has a ton of built-in UML diagram types.
- Threat and failure modeling are far easier to get right by reusing component relationship diagrams that you should already have. Use Excel spreadsheets to categorize and rate threats and failures and to specify mitigations.

Eric Aside We now use a threat-modeling tool to do this work. You can find some on MSDN.

Details, details

Just as each step shown in the preceding table drove the next clockwise turn around the design dimensions at a high level, the same process occurs at more detailed levels. You basically spiral into implementation through the following steps, which are closer to the center of the table:

- **Use cases** Simple descriptions of how actors use the system to perform tasks. Visio even has built-in shapes for use cases if you like pictures.

  Eric Aside The term *use cases* is often used to describe very high-level scenarios, not the simple low-level ones I’m talking about here. Sorry for any confusion.

- **API definition** When you have the use cases, specifying the application programming interface (API) correctly becomes far easier. This step solidifies the contract between components and objects. Then you can actually start coding the implementation.
- **Test-Driven Development** TDD provides an ideal framework for methodically and robustly creating a simple, cohesive, and well-factored implementation, complete with a full suite of unit tests.
- **Sequence diagrams** For particularly complex functions or methods, use sequence diagrams to clarify code constructs.

A few comments on the whole design process I just outlined:

- The process has no unnecessary complexity or added steps. Each step is well defined and leads into the next step with all the information needed.

- As with any process, you may be tempted to skip steps to save time (like skipping straight to TDD without a PM spec, dev spec, or use cases). That's fine if the outcomes of those steps are trivial, but you invite peril if the outcomes are tricky (and around here outcomes tend to be tricky).
- You know what enough is and when you are done.

Show me what you're made of

Okay, so I glossed over two bits:

- How do you get the right scenarios and personas?
- How do you bridge the gap between a PM spec built from those scenarios and personas to a dev spec with a class hierarchy and relationships and a component stack and relationships?

There are two ways to get the right scenarios and personas. One way is through direct contact with customers. The second way is top-down, starting with the following even higher-level steps in the design process spiral:

- Market opportunities and personas (external-dynamic)
- Product vision document (external-static)
- Product architecture (internal-static)
- Subsystem interaction diagrams or flow charts (internal-dynamic)

Then continue spiraling into the scenarios and personas as discussed previously.

Eric Aside When you've got hundreds of millions of customers, thousands of engineers, and billions of dollars in the balance, I'd advise using a rigorous approach to high-level design. If that upsets you, I'd advise staying away from large, successful projects. When that much money is involved, projects are driven by politics or process.

Mind the gap

Bridging the gap between a PM spec (or higher-level product vision) and a dev spec (or higher-level product architecture) requires mapping functional requirements (like features) to design parameters (like classes or components). There are two straightforward methods for doing this:

- **Design patterns** Recognize that you've solved this problem before, and reuse the old design.

- **Axiomatic design** Use this method for brand-new designs and even for cleaning up and evolving old designs. The steps are fairly clear-cut:

  1. Create a table in Excel or Word, with the rows listing the functional requirements and the columns listing the design parameters. The internal cells should be blank to form a matrix. This is known as the design matrix.
  2. Ensure that each functional requirement is written to be orthogonal to the others. This often means breaking up complex requirements into basic pieces. Adding new requirements just means adding new rows (nice).
  3. One by one, list design parameters that would satisfy each functional requirement. Often a design parameter (a class or component) helps satisfy more than one requirement. Mark an X in each cell under the design parameter column that helps satisfy the corresponding functional requirement. The design matrix should now have Xs scattered throughout the cells.

     Okay, so by now we need an example. A simple one is the design for a faucet. The two orthogonal functional requirements are to control flow and temperature. Consider two alternative designs, each with different design parameters: separate hot and cold water taps, and a single lever with tilt controlling flow and rotation controlling temperature. The two corresponding design matrices look like this:

     Two alternative design matrices for a faucet

	Hot tap	Cold tap		Tilt lever	Rotate lever
Flow	X	X	Flow	X	
Temperature	X	X	Temperature		X

  4. Rearrange the design parameters so that no Xs appear above the diagonal of the design matrix. You may need to rethink how classes or components can better match to a smaller number of requirements to get this to work.

     Notice in our example that the separate hot and cold water taps have Xs above the diagonal, while the single-lever design is nicely factored into a diagonal design matrix. The single-lever design produces a far better faucet.
  5. The resulting design matrix will document a nicely factored design. If all the Xs are on the diagonal, the design is completely decoupled and will be a cinch to implement. Any Xs below the diagonal indicate dependencies and can help you guide the proper order of development. Any Xs left above the diagonal that you couldn't remove are nasty, cyclical dependencies that should be handled with great care.

Your recipe for success

So there you are—a step-by-step guide to minimal, yet complete and robust design. It may seem a bit daunting, but really each step is straightforward. By using a methodical approach, you won't miss anything, you can schedule it, and life won't seem quite so open-ended when the pressure is on.

This type of complete approach also cures analysis paralysis. Follow every step and you're done. If people try to add extra steps or bring in extra requirements, you've got documentation to halt their advances.

As I mentioned in my last column, "A software odyssey—From craft to engineering" (which appears in Chapter 5), dropping bug counts by a factor of a thousand was a necessary step toward meeting the quality that our customers are demanding. The other is discerning the requirements and creating a detailed design that meets them. By using a complete design approach along with an engineering-based implementation, we can exceed our customer's expectations for quality and value, turn around our industry, and stomp our competitors in the process.

February 1, 2006: "The other side of quality—Designers and architects"

Quality. Value. Motherhood. Ideals we can all agree are good things. Not too many people argue against them. But what is quality? What is value? I'd ask about motherhood, but my wife claims I'll never understand.

I've been at offsites where we've been asked to define quality and value. It's offsites like these that make hellfire and brimstone sound appealing. Of course, we get nowhere. If it weren't for the free chocolate chip cookies, the entire time would be wasted.

The trouble is that quality and value are wrapped in perception. Even if we built a product that precisely matched the spec with zero bugs, customers and reviewers might still hate it. If it doesn't work the way they think it should, it's junk. If it doesn't solve the problem the way they expect, it's trash.

At the same time, I-Puke can produce a buggy piece of glorified plastic that crashes twice a day, and it will get showered with praise and command premium pricing just because it reminds people of their pet rocks. Life isn't fair. The market isn't fair. And customers are fickle.

You'll have to do better than that

Of course, whining about unfair markets and fickle customers is pathetic. Microsoft has traditionally dealt with customer capriciousness in two ways: wowing customers with new must-have features and chasing down competitors.

Unfortunately, as the software market matures and broadens, customers get featured-out and computer usage becomes more critical, making customers more conservative. These days, customers often equate more features with more headaches. "Make the features we already have work" is the kind of enigmatic advice customers offer.

Even chasing down competitors has its limits. The strategy works well because you know exactly what customers want. Unfortunately, when you catch up, you are back to being clueless. You have jumped ahead only to fall back behind.

A change would do you good

How do we break the cycle? That's easy—with designers and architects. By designers, I mean the people involved in defining the user experience—the "what." By architects, I mean the people involved in defining the end-to-end implementation approach—the "how." Customer and business needs define the "why" and "when."

There are two sides to quality: design and execution. At Microsoft, we execute like nobody's business. Though we still have a way to go, we are improving the quality of our execution all the time.

Eric Aside The insight about quality being a combination of experience design and engineering execution is another key piece of wisdom I've gained over the years. Too often people don't separate the two or think about only the experience or the engineering.

But what good is perfect execution if you produce a product nobody cares about? You need great design, and that's where designers and architects rule.

Good designers really understand the entire customer experience. They think it through, end to end, and design a solution that looks beyond hardware, software, networking, and other technical boundaries. The result focuses simply on how to thrill the customer. Key aspects for designers are simplicity, seamlessness, key scenarios, differentiation from current solutions, customer-driven constraints, and perceived value.

Good designers produce a pipe dream. Good architects make it real.

The man just got it wrong

Many developers think architects are all about diagrams, abstractions, and deep thoughts, instead of practicality, efficiency, and execution. They've met architects who concentrate on purity and elegance instead of being down and dirty in the code. There's a name for smug, detached architects: unemployed.

Good architects have a firm grip on reality. It's what differentiates them from designers. Though, the best designers and architects can dream beyond boundaries, yet live within them. Architects break down the pipe dream created by designers into a variety of implementation options. Each option has its own advantages and issues.

Doing it well

Good architects keep all their options open for as long as possible. As they think through how to translate the current state of technology into the designer's future vision, constraints will arise that restrict their options—constraints like cost, performance, security, reliability, legal considerations, partnerships, and dependencies. Eventually, few options will remain and an optimal, high-level implementation design—AKA the architecture—will emerge.

The next task for architects is to clearly articulate the architecture to all the products and components that together realize the designer's vision. The two most important factors are the interfaces and the dependency graph between the pieces. Often existing components and interfaces must be refactored to avoid circular dependencies or enable new variations of behavior.

Naturally, the lowest-level dependencies must be the most stable and developed first. Part of the job for architects is to devise practical ways of accomplishing this. Often it means first updating the interfaces and leaving them stable while the code underneath is reformed.

Eric Aside At the risk of sounding conceited, I'd say the last three paragraphs are worth another read. They cover the simple steps that great architects follow yet most architects miss.

Next time, try sculpturing

Regardless, before a single line of shipping code is written, the designers and architects must envision and describe the road map to realize customer value. Oh, I can hear naysaying nitwits crying "Waterfall!" at even the hint of thinking before acting. But the idea is to create a road map, not a full-scale replica. By knowing what you want to achieve and thinking through the pitfalls, you might actually have a chance of shipping a real solution.

Because the road map isn't the real thing, there's still plenty of work for the product teams that are fleshing out and executing the design. What do the designers and architects do during that time? Jump to the next project? Draw more pictures? Present papers at conferences? No, ding-a-lings—that would be disastrous.

Designers and architects must keep the product teams honest and resolve all the conflicts that arise. No one has a big enough brain to think of everything, nor should you waste years trying to do so. Good designers and architects leverage what they know as a foundation and then work with the product teams to resolve the remaining issues as answers emerge.

This means designers and architects must have strong communication skills. They must work well together as well as with the product teams to present a consistent message and guidelines. They must also have large enough brains to keep the big picture in mind as they review the details that make it possible.

Just the right tool

In addition to their vision of the future, the key tools designers use for keeping product teams on track are end-to-end scenarios and critical-to-quality (CTQ) measures. While the practice of defining and testing end-to-end scenarios is gaining critical mass support at the company, CTQ measures are just taking hold. These are the small handful of measures that will make or break the customer's sense of quality and value. They could be performance or reliability measures, like time to complete a task or uninterrupted connectivity. They could be usability features like only authenticating once or consistency of interface. Whatever speaks "quality" to the customer the loudest, those are the CTQ measures.

In addition to the architecture, the key tools architects use for keeping product teams on track are the interfaces and the dependency graph. (The dependency graph could be as simple as a block diagram or as complex as a wall-sized component graph.) The whole idea is to allow hundreds of small teams to work independently while still creating a seamless experience. The pieces will never fit without the right interfaces. The work will never remain independent without respect for clear boundaries between layers.

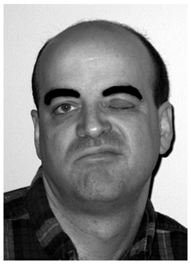
Eric Aside The whole idea of small team independence was important enough that I devoted the next column in this chapter to it, "Blessed isolation—Better design."

Beyond these walls

Designers and architects play different roles, but they share one aspect in common: broad, end-to-end scope. Their job is to think across feature boundaries, product boundaries, and even market boundaries to seize opportunities. That's because customers don't see any boundaries. They just see incomprehensible, unmet potential. Designers ignore the artificial boundaries to bring out the value. Architects wrestle the boundaries into submission to make it real.

Sure, we could continue to frustrate customers, play catch-up, or blow years of a lead messing around until we realize that nothing fits together. Or we could take advantage of our breadth and depth to deliver unparalleled and uncompromising value to our customers. It's not hard, but it does require the discipline to think broadly before we act, then the focus to follow through. Achieve that, and we will finally beat our toughest competitor in the market: ourselves.

August 1, 2006: "Blessed isolation—Better design"



I received plenty of what I'll call "feedback" on my last column, "Stop writing specs, co-located feature crews" (which appears in Chapter 3). According to my critics, design documentation forces design to occur and differentiates a robust architecture from, well, much of our code. Somehow readers got the impression that I was against documented design. Perhaps it was the column's title.

That impression is wrong. I am for design; I am against waste. I believe if you work with a small multidisciplinary team in a shared collaborative space, on one feature set at time from beginning to end, then you don't need to write formal specs. Continuing to write formal specs in such an environment is wasteful and should stop.

But what if your feature set is so big that your feature team has more than eight people? What if your team is spread across floors or even continents? What if you first design groups of features, then implement them all, and then test them all? All these cases call for formal written specs. Without them you can't keep your team on the same page over time or distance.

Breaking up is hard to do

Architecture. Now I can hear naïve naysayers nitpicking, "Then your whole column was pointless. No real product feature is so small or independent that it can be done by fewer than eight people in isolation." Architecture. "You're just like those Agile extremists," the dogmatic dinosaurs would continue. "They talk a good game, but at Microsoft we work on big products for big customers with big issues." Architecture.

Is there a way to bridge the gap between big products and small teams? Why yes, there is. It's called, "Architecture." The whole point of architecture is to break intractable problems into little manageable ones. Architecture, when done properly, delivers the needed isolation.

Doing it well

Unfortunately, there are countless examples of bad architecture and bad architects out there. How do you do architecture right? Funny you should ask.

While creating robust, reliable, and resilient architectures is hard, the process itself is simple:

1. Collect scenarios and requirements the product architecture must satisfy.
2. Ensure those scenarios and requirements are clear, complete, and independent.
3. Map scenarios and requirements to product components that will implement them.
4. Identify interfaces between the product components.
5. Inspect the components and interfaces for failures and vulnerabilities.
6. Document the components and interfaces.
7. Redesign and refactor as issues and new requirements emerge.

Before I get into the details, which hold all the entertainment value, let’s first talk logistics.

There is no “I” in team

I’ve known architects whose lives are out of control. Because architecture is broad by its very nature, there are loads of people involved. Some architects spend their entire days in meetings with stakeholders, followed by nights and weekends doing the actual architecture work. While the commitment is impressive, it’s not advisable and likely unsustainable.

Of course, it’s no better for an architect to work alone and ignore stakeholders. A far better model is the architecture team.

The architecture team is composed of senior engineers from each feature team and chaired by the product architect. They meet regularly, as often as daily at the beginning of a project, and no less than weekly thereafter.

The architecture team as a group performs all the steps I listed. One member is responsible for documenting the information collected and decisions made. (That role can rotate.) Instead of the architect traveling to stakeholders, all but the most senior stakeholders come to the team. This vastly reduces travel time and randomization for the architect.

Some product lines have architecture teams composed of product architects and chaired by the product line architect. This hierarchy works well to cover the most complex cross-product scenarios and requirements.

Step by step

Okay, we've got the steps and the people to do them. Now how about those details:

- **Collect scenarios and requirements the product architecture must satisfy.** This one should be a no-brainer. The key is getting stakeholders to review these areas with the architecture team as needed. Remember to consider the impact of quality requirements (like security), as well as functional requirements.
- **Ensure those scenarios and requirements are clear, complete, and separable.** Getting clear and complete requirements puts a burden on the architecture team, but should be clear-cut. (Don't forget performance.) The separable piece is more subtle. You want separable requirements to avoid circular dependencies. However, often multiple scenarios relate to the same underlying requirement. In that case, the architecture team must break up the scenarios into the shared piece and the independent pieces. The shared piece becomes a separate requirement.
- **Map scenarios and requirements to product components that will implement them.** This defines the architecture, the layering of the components, and their responsibilities. Ideally each requirement should be implemented by one and only one component. That would provide pure isolation and make implementation a snap.

  In real life, many components are often involved in many requirements. The components involved in the largest number of requirements are your biggest dependencies. They need to be the most stable and get completed first. Circular dependencies are to be avoided at all costs. Axiomatic design can be a useful tool for doing this difficult mapping step.
- **Identify interfaces between the product components.** Once you have the mapping, identifying interfaces is easier. If two components work together to implement a requirement, they need an interface between them; otherwise, they don't. Generally speaking, only requirement-driven interfaces should be defined at the architecture level.
- **Inspect the components and interfaces for failures and vulnerabilities.** Quality issues around security, reliability, manageability, and so on are often apparent directly in the architecture. Spotting and resolving them or mitigating them up front can save you countless hours later.
- **Document the components and interfaces.** Yes, I said it. You should write formal documentation for your architecture. Even if your product group works with small multidisciplinary teams in shared collaborative spaces on one feature set at time from beginning to end, those small teams require isolation to function independently. They get that isolation from clearly defined interfaces and component responsibilities. The architecture is shared across teams, time, and distance, so it must be documented.

Once the component responsibilities and interfaces have been debated, designed, debugged, and documented, the architecture is done. It's done, but it's not finished. The architecture doesn't truly get fleshed out till all the feature teams implement it.

- **Redesign and refactor as issues and new requirements emerge.** Regardless of how smart or thorough your architecture team is, they will always miss issues and get blind-sided by new requirements. That's why it's important for them to keep meeting every week. Problems that arise get brought before the architecture team, which returns to first principles and refactors the design as needed. The architecture team consists of senior engineers from each feature team, so the problems will be visible and well understood.

Eric Aside The C# architecture team, led by Anders Hejlsberg, still meets for two hours three times a week, even though the original spec was completed eleven years ago. While the number of team members has varied, Anders says it ideally consists of six people plus one PM who takes notes and keeps the agenda.

Dogs and cats living together

Once the architecture is documented and the components are isolated, individual feature teams can engage free from conflict. Should an unforeseen problem arise between components, the architecture team can quickly address it.

There's still plenty of bottom-up design left to do. Because the scope is smaller and the impact confined, there's ample room to experiment and apply less burdensome Agile design techniques, like Test-Driven Development.

Were you to attempt bottom-up design for the whole product, the simultaneous number of conflicts requiring broad redesign would quickly collapse the project under its own mass hysteria. It's one thing to rethink a single component behind a stable interface. It's another to constantly change interfaces and responsibilities across a million-line code base.

By using top-down design just enough to provide isolation and bottom-up design enough to optimize collaboration and quality, you get the best of both worlds. By using architecture teams, you coordinate effort, create growth opportunities for your senior engineers, and achieve blessed isolation. It's a beautiful, peaceful world within your reach.

November 1, 2007: "Software performance: What are you waiting for?"

You hurt your shoulder playing volleyball, so you make an appointment to see your doctor. You enter the office and wait in line for 5 minutes just to let the receptionist know you've arrived. He has you verify your contact and insurance information, which haven't changed in ages, and then tells you to sit in the waiting room.

You sit in the waiting room for 10 minutes, inhaling all kinds of ailments from the crowd, seething about how you're going to leave sicker than you were when you came in, till a nurse shows you to an exam room.

After 5 minutes in the exam room, another nurse comes in, takes your vital signs, and has you repeat the reason you gave for the appointment when you originally made it. Ten minutes later, your doctor arrives and actually addresses your shoulder injury.

Welcome to a user's experience running our software. You wait forever just for it to launch. You provide your credentials again, even though you gave them when you logged in. You wait again for your personalized environment to load. You click a few menu items or buttons to launch the specific functionality you want. Finally, you wait again while the feature prepares to do what you actually launched the software to do in the first place. That's assuming there aren't network delays.

Just one moment, please

Waiting is dull. Waiting is frustrating. Waiting is agonizing. Waiting is just an unbelievably bad time all around, on every level. Nobody likes to wait. Nobody asks to wait. So, why the heck do we make people wait?

Actually, why do our customers even put up with it? Why do I put up with it at my doctor's office? I guess because all doctors' offices are slow. But if one of my friends told me about a quick doctor's office that provided comparable care, I'd switch in a heartbeat.

This means a competitor's quicker program could flatline our business. How do we make our programs quick, before our competitors do? I'm glad you asked.

Eric Aside If your doctor is good enough, you may put up with significant hardship instead of switching. However, giving people a reason to switch is inexcusable, especially when it's easy to do better.

You're faster than this

So, you want better performance from your software. Where do you start? "I know, I know!" says our resident performance pariah, Mr. Speedy. "Profile your code, find out where it's spending all its time, and then optimize, perhaps even parallelize, those inner loops."

Well, Mr. Speedy, aren't you clever. Let's profile our doctor's office, shall we? Whoa! It turns out the doctor is always busy, and that's the bottleneck. Who would have guessed? According to Mr. Speedy, all we need to do is speed up the doctor, find a faster doctor, or get two doctors to do the job of one. Right? Wrong!

There's nothing wrong with my algorithm—I mean, my doctor. If you made her faster, she wouldn't be any better. In fact, she'd probably be worse. Doing the job right takes time, and making all kinds of optimizations might improve things a little, but it might also cause mistakes. I don't want a different doctor who happens to be faster, either. I like my doctor. I know her well, and she knows me.

I also don't want two doctors. Even if they are twins, I'll never know which thread—I mean, which doctor—I'll get. They might both try to treat me. They'll have to communicate with each other all the time to avoid mistakes. They might even get stuck waiting on each other. It's way more complicated, and it really doesn't solve the problem even if two doctors are twice as fast. I've still got to deal with the receptionist, waiting room, exam room, and nurses.

Eric Aside "But what about multicore processors?" you might ask. Look, there's using technology for the sake of technology, and there's using technology for a purpose. If the user experience demands threading across multiple processors, I'm all for it. If not, you're just giving yourself a cheap thrill at the expense of the customer.

Should I keep a copy?

"Hold on," says Mr. Speedy. "What you need is a cache—that always speeds things up." Hello! We've got cache fever in the doctor's office. That's part of what's slowing us down. We've got a reception line cache, a waiting room cache, and an exam room cache.

It seems like everyone at the doctor's office is concerned about speeding up their own work, so they all created their own caches. The receptionist created a cache, the nurses created a cache, and the doctor created a cache. The result is that patients spend all their time waiting and moving between caches instead of being processed by the doctor.

Think this doesn't happen in code? You've obviously never looked inside those database, shell, and system calls you use. All that data you're caching for "performance" is already being cached for the same reason by those functions. Sometimes there are as many caches as there are layers. Every cache has a fetch and memory cost to it. Well, I'm cached out.

You’re not being the ball

Let’s start over, shall we? Instead of speeding up the existing doctor’s office, as marginally effective as that might be, let’s think about things from the patient’s perspective. What would you and I, as patients, like the experience to be?

Here’s what I’d like. Check my contact info and insurance when I call in for an appointment. Write down my symptoms and include them in the appointment. Heck, let me do it all online (wait, that’s crazy talk)!

When I show up at the doctor’s office, I can walk right to my exam room (just one level of caching). It’s the room with my name above it, just like at the good rental car agencies. A big sign in the room says, “Please take off your shirt and have positive identification ready for the nurse, then hit the big ‘I’m ready’ button.”

The nurse, seeing that I’ve hit the “I’m ready” button, comes in, checks my ID, takes my vitals, and hits the “Vitals taken” button, which adds me to my doctor’s queue. As soon as my doctor’s available, she comes in and addresses my needs. That would be great! Heck, there could even be a monitor in the exam room with queue stats and predicted wait times. The stats could be used to fine-tune the number of appointments available per hour to minimize wait times, while still ensuring that each doctor is fully utilized.

Eric Aside If you haven’t read about the Theory of Constraints or its drum-buffer-rope approach to optimizing results, you are in for such a treat. They should be required reading for anyone trying to rethink and revolutionize performance of everything from software to cafeterias.

Have you ever been experienced?

There, that wasn’t so hard. Setting up a doctor’s office the way I described would be easy and not that expensive. It doesn’t require more doctors or faster doctors, and it actually saves floor space. Sure, the online appointments and predicted wait time monitors would require special software, but those aren’t essential to get a better and faster experience. What is essential is to think through the customer’s experience with a view toward minimizing wait time.

Here are some questions for your consideration:

- When was the last time your team thought through the end-to-end customer experience, including the wait time?
- How would the customer want to deal with the inevitable constraints that every process has, besides giving them a CANCEL button? (Associating our software and services with “cancel” seems unwise.)

- How could you minimize the impact of errors, network delays, and device I/O in a way that customers would find natural and unobtrusive?
- What measures and statistics could you use to fine-tune the experience, minimizing wait times while getting full utilization of key resources?

Right now we design experiences as if these performance constraints don't exist. Everything is modal and synchronous, as if functions always return and people never select the wrong option. We design a feature at a time instead of end to end; or if we do design end to end, we think only about the ideal scenarios, not the likely ones. We assume exceptions and delays are unusual, even exceptional. That's naïve, which is a kind way of saying "stupid."

Eric Aside Of course, there are great examples in Microsoft software for thinking through the end-to-end experience quite nicely. I mention some in the next section.

You'll be ready

Performance tuning does have its place. There are functions and services that must scale up and out. There are issues with blocking and locking that require special care, which a real performance expert can help you resolve. It's just that those aren't the common case.

The common case is the ordinary case. A customer is trying to get something done. It involves network access and I/O. Those interactions could fail or cause delays. The customer generally has experienced these problems and knows they exist. The best way to handle them is to talk to customers and understand ideally what customers want to happen.

Perhaps the customer would be happy if the I/O completed asynchronously—the solution Outlook and OneNote use to vastly improve the customer experience. Perhaps the customer would be happy to work on a local copy and synchronize on demand—the solution ActiveSync and FrontPage use. Perhaps the customer is happy to queue their requests and get a status report later—the solution build systems and test harnesses use.

What about me?

The key is to look at the world from the customer's perspective and to design an experience that anticipates failure modes and minimizes their impact on users. Performance should be specified in the experience with specific measures and guidelines, not left to chance or hope. It typically doesn't require complex algorithms or fancy caching, both of which can be overdone. It requires being thoughtful and deliberate.

When performance is specified through the experience, it's built in and tested from the start. No one gets surprised or has to scramble at the end of the project to suddenly become a performance expert. The only surprise is on the customer's face when what they thought would be an agonizing doctor visit turns out to be a delightful one.

April 1, 2008: "At your service"

Remember this one, "The microprocessor changes everything!" No, it didn't. Yes, it had a big impact, but people still fretted about the same problems and tried to accomplish the same things. They just created problems and accomplished things more efficiently. How about "The Internet changes everything!" No, it didn't. Yes, it had a big impact, but people just got that much better at creating the same problems and accomplishing the same things. Now we have "Software plus services changes everything!" Oh, spare me.

You might claim, "I. M., you're wrong about this one. Microsoft did change dramatically with each of those shifts." We changed quite a bit, but not dramatically. If we had changed dramatically, we would have cut or replaced the majority of the workforce (talk to your friends at other companies). Instead, we augmented our tools and our workforce to take advantage of each new opportunity.

Look, I don't want to downplay the importance of any of these shifts. They improved people's lives and made the world smaller and more accessible. They grew new businesses, improved quality, and increased productivity. That's fantastic. But don't go around saying a new technology changes everything, because the things that count don't change: people, their problems, and what they want to accomplish. Just ask those companies who focused on the technology instead of the customer. Oh wait, you can't. They're bankrupt.

Eric Aside Naturally, the shift to the microprocessor led to establishing Microsoft. So, I suppose that does count as a dramatic change for the company. However, my point remains the same—everything didn't change. People didn't change. They still needed to maintain relationships, earn a living, and achieve personal and business goals. What changed is how people accomplished these activities.

I'm fuzzy on the whole good/bad thing

This is why nothing makes me more miffed than engineers who say, "What is software plus services?" Or much worse, "It's a new world; we've got to create services!" Hearing that makes my lunch call for a reappearance. Stop focusing on the technology you fools! It's not about the technology. It never has been. It's about the customer and what they want to do.

Hold on, I hear an Ozzite calling: "But Ray says it's all about services and the cloud." Listen, when Ray Ozzie talks about services and the cloud, he talks about things people want to do and how services and the cloud can help. The starting place is always the same: customers and their goals.

"Oh, so we focus on how customers want to use our software with services, right?" No! Get this straight! Customers don't want to use our software for the sake of using our software, in case your family and friends haven't made that clear. Customers want to accomplish goals, like writing a term paper, sharing pictures of their kids, or fulfilling a purchase order. To the extent that services help customers achieve their goals, we want to integrate and provide those services.

Perhaps some examples would help. Let's talk more about that term paper, the pictures of the kids, and that purchase order.

On good terms

Writing a term paper is a common enough goal for students that businesses are built around supplying term papers for a fee. If we want to help students write term papers, the first step is imagining what a wonderful term paper experience would be like. We've all had horrible term paper experiences, so this shouldn't be too hard.

Okay, so the instructor assigns students to write a paper about preparations for the Beijing Olympics in 2008. Stu, our student, opens a word processor and starts jotting down a rough outline. "Let's see," says Stu, "I'll start by talking about the Olympic Games, then the city selection process, then perhaps a history of Beijing, followed by some details of Beijing's bid, followed by their preparation plans, and closing with the current status." Nice outline, Stu.

Stu hits the research button and phrases in the outline get highlighted. Stu clicks on *Olympic Games* and a search result frame pops up with info about the games. Stu scrolls through previews, selects the ones he wants, and the text is inserted directly into his document. He repeats the process for city selection, Beijing, and the other sections. In each case, the search is narrowed by the context of the previous searches, knowing Stu cares about the Olympic Games.

Next, Stu edits the text to make it sound more like he wrote it and actually understood it. The grammar checker, in addition to highlighting grammar and spelling mistakes, also highlights sections that are too close to the source text to be considered original. When the text is clean of errors and outright plagiarism, Stu sends it to his instructor.

I wish my term papers could have been that easy. So, what pieces of this scenario does Microsoft have already? We've got a word processor; we've got Internet search; we've got a grammar and spelling checker; and we've got e-mail. What we need is a plagiarism checker and a client user interface for Word to talk to Live Search and download parameterized search results. That shouldn't be too hard. Voilà, software plus services that is actually valuable.

Sure, it would be even more valuable if Stu could store his term paper in the cloud as he worked on it from multiple devices, including his car and phone, but you can't start there because the cloud and phone are cool. You start with Stu and what Stu wants to do.

Eric Aside A subtle point here is that perhaps you don't want to make writing term papers so easy. They should be more about discovery and integration of ideas rather than web searches and editing. Checking against plagiarism helps, but it has become a bit too easy to pull term papers together. Perhaps it's time to move the end-to-end scenario back to the instructor creating a challenging and instructive assignment. I'll leave that as an exercise.

Preserve your memories

We all know the sharing-pictures scenario. You plug in your camera, upload your pictures, and click the button that should be there to share pictures. Okay, on Windows XP there's a link to **Publish the selected items to the Web**, and on Windows Vista there's a **Share** button, but it doesn't share across the web. I get why this scenario is hard when it comes to open markets and working with partners. That's a good reason, but not a good excuse.

Even in the Windows XP case, assuming the customer can find and understand the **Publish the selected items to the Web** link, have you tried it? It focuses on giving Windows XP the information it needs to complete the operation, not on the customer trying to share their pictures. This is a case where we've got just about every component we need to deliver a useful software plus services scenario, but we've designed the experience to serve our software instead of serving the customer.

Eric Aside I don't mean to demean how incredibly difficult it was to create the picture-sharing scenarios we have. The Windows XP work was an early attempt at handling a complex web of handoffs and agreements. Governments pressure Microsoft to avoid integrating too much into the operating system. Partners often don't agree or share with each other. The key is not letting those considerations influence the design of the end-to-end customer experience. Once we have that compelling story, we can then consider the practical constraints.

Self-fulfilling prophesy

Most of us shop online for at least some of our needs these days. While it's tempting to talk about the end-customer buying experience, let's talk instead about the order fulfillment experience of the online retailer.

When a customer submits their order, the retailer must:

- Receive validation of the credit card from the bank.
- Bring up the confirmation page and e-mail a copy to the customer.

- Check inventory for the product.
    - If the product is in inventory, send the instructions to the warehouse to ship the product to the customer and update the inventory.
    - If the product isn't in inventory, order the product from the supplier with instructions around labeling and routing the product to the customer, and then track and record the transaction for data mining future buying.
- E-mail the shipping information to the customer.

All this seems simple enough. Once again, we've got most of the components—web services for the transactions, ASP.NET for the confirmation page, Exchange for the e-mail, SQL Server for the inventory, and Dynamics for all the business logic. The key is, "Does this scenario come together as easily as it sounds?" When our small business customers try it, do they have to fight our software the whole way?

Remember, customers believe that all Microsoft software comes from one place and that Bill Gates writes all the code. They don't understand why one application works differently than another. They can't afford to be experts at setting up each different piece. Making software plus services work isn't about clever new code. It's about focusing on what the customer is trying to accomplish and making that easy as it crosses boundaries.

Eric Aside Enterprise customers often want to design the end-to-end solutions themselves, or through consultants who understand the special details of their business. However, we still need to design for those end-to-end scenarios. The faster and easier it is for enterprise customers and their partners to realize their scenarios, the better their results will be, the less costly they'll be to maintain, and the more they'll appreciate our work.

We can get together

Okay, you're right; there are aspects of software plus services that involve change. The service piece is not a static image on a DVD. It's a living and breathing entity in a datacenter whose life only begins the day you ship. When you patch, upgrade, or revert a service, it all has to happen while the service is still live and functioning. Debugging is insanely difficult, so you need to instrument your service to diagnose problems across machines from anywhere in the world.

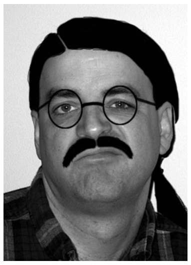
All this implies designing your service, and the software that uses it, from the ground up to be resilient, maintainable, and manageable. You don't get to walk away from services after the ship party; the party is just beginning, as is your commitment to keep improving your service every day.

However, none of that matters if we get seduced by the technology and features and lose sight of the customer. The customer doesn't want software plus services. The customer wants to achieve goals, and well-designed software plus services can help.

The real trick to software plus services is working well across product and group boundaries to make end-to-end customer scenarios work the way the customer expects, instead of the way our company is organized. Learn to do that well, and you will change everything.

Eric Aside You can read more about how to work well across product and group boundaries for end-to-end scenarios in my column "The other side of quality—Designers and architects" earlier in this chapter.

August 1, 2008: "My experiment worked! (Prototyping)"



It's summertime. Time to sit out in the sun and daydream, perhaps on a vacation or a weekend afternoon. When your mind is relaxed at times like these, you often think of beautiful new ideas. You further develop those ideas and then, when the time is right, perhaps early in the next release cycle, you begin prototyping those beautiful notions. Before you know it, your beautiful ideas have blossomed into hideous, miserable nightmares that either die of exposure or, worse, live on to cause future generations of engineers to curse the day you were born.

Oh, but if it were a fairy tale. Instead, more often than not, prototypes of beautiful ideas become horrific, hairy hodgepodges of hacks that cannot be easily maintained, refactored, or understood. Why? What happened?

It's not that you should write prototypes more carefully, with unit tests and all the rest—you shouldn't. It's not that you should throw the prototypes away—though you should. No, the problem is that your entire philosophy about prototyping is dead wrong.

Explore the space

Usually, when engineers think of a new idea to try, they write a prototype. That's a major mistake and the wrong thing to do. It leads you down a path to destruction of all that was good about your idea. You see, you shouldn't write a prototype—you should write dozens of prototypes parameterized to try hundreds of cases, all designed to solve the same problem but from different angles.

That's how all other fields of study work. You don't do one experiment, try one approach, or use just your first guess. You do hundreds of experiments. Artists and producers call it "exploring the space." Could you imagine if medical researchers tried only one idea at a time to cure diseases? Wouldn't you think that was idiotic? Hello?

That’s so rad!

Naturally, you don’t have all the time in the world to write dozens of sophisticated prototypes. Good, you’re not supposed to. Prototyping isn’t supposed to be like production engineering. It’s supposed to be like experimenting. It’s supposed to involve the software equivalents of duct tape, silly putty, and wire hangers, like VBScript, WordArt, and Perl. You’re supposed to throw together prototypes in hours not days.

If it’s taking you longer to write one prototype than it did to understand the problem, you are already off the end of the gangplank and are headed into the shark-infested waters of failure and frustration. Spending serious time writing one prototype not only distracts you from exploring other possibilities, but it also puts so much investment in that one prototype that you can’t help but use it as the basis for production code. Welcome to despair and desolation.

On the contrary, prototypes should be churned out so fast and furiously that no one would consider taking them seriously as the real thing. Prototypes are there to try out ideas, make mistakes, iterate, and gain insight. If you write only one of them, then you’ve learned nothing, aside from the fact that you are a close-minded, ignorant, misguided fool.

Harness in the good energy

Hold on, the “I can’t change” people are calling. They say “You can’t prototype quickly without frameworks, harnesses, and libraries. It takes time to build those tools. Creating dozens of prototypes with our shipping pressures just isn’t realistic.” Look, if you are consigned to being a calcified carcass of a coder just admit it and accept it.

Otherwise, you need to wake up and realize that prototypes don’t have to follow the same rules as production code. They can be written in different languages and be built on different platforms. They can use tricks and shortcuts, scripts and canned animation; you name the kluge, short of stealing licensed code, and it’s fair game for prototypes.

There are loads of libraries, heaps of harnesses, and enough frameworks to fill a football field to help with rapid prototype development. All you need to do is step out of the box and into the garden.

You still have a choice

Great design, like great research, depends on trying dozens of ideas. It also relies on collaboration, working with others to generate new ideas and directions. If there’s a user interface (UI) involved, you want user experience (UX) folks to help inspire you. If it’s a library or API, you want architects or clients as your coconspirators.

Eric Aside UX engineers are designers and usability experts. These folks are trained in creative design thinking, thus the perfect people for prototyping projects. Many even specialize in the design and usability of interfaces, making them ideal to help with API and library design.

Once you've generated dozens of ideas, prototyped them all, iterated, and gained the desired insight, you've got a number of options. You can:

- Pick your subjective favorite.
- Use a simple tool like a Pugh Concept Selection Matrix to thoughtfully choose between alternatives.
- Combine aspects of each approach and experiment some more.
- Postpone making a decision till constraints force a choice.

Postponing design decisions till constraints force your hand is called *set-based design*. You keep all design options available, culling ideas only when they break a requirement. Because schedule requirements often hit before all alternatives have been considered, you eventually need to fall back on something like Pugh Concept Selection. Until you do, set-based design keeps your mind open and helps you find the optimal solution.

Eric Aside Pugh Concept Selection uses a simple table to rate alternatives against requirements. Ratings are positive, negative, or zero based on fit as compared to your default choice. Requirements are scaled according to significance. The alternative with the highest total wins.

Online Materials You'll find a sample spreadsheet (Pugh Concept Selection Example.xlsx) in the online materials.

Throwing it all away

By the time you've made your design choice, your duct tape and bailing wire prototypes should be so fragile and convoluted that no one would consider keeping them for anything but nostalgia. That's the right result.

Using cobbled-together code as the basis of production work only leads to software that is difficult to maintain, susceptible to failure, and must be retrofitted to meet quality requirements such as automated tests, code reviews, globalization, accessibility, security, privacy, manageability, and performance—just to name a few. The result of this retrofitting is the hideous hodgepodge of hacks I mentioned earlier. Not the beautiful ideas that started you down this path.

Throw away your prototypes, using them only for idea and algorithm reference, not code. This shouldn't be painful, because you spent so little time on each.

Temptations always come along

I know there is a great temptation to write solid prototype code using quality methods, just as there is a great temptation to write rapid production code using quick-and-dirty methods. You must avoid these temptations by any means necessary.

How you act and code during experimentation should not match how you act and code during production. People who don't understand this difference have a name. They are called "children."

If you are treating prototypes like production code, or vice versa, you should get reamed on your review. If your boss thinks you should write prototypes like production code, or write production code like prototypes, he should be fired. Not that I feel strongly about it.

The result of production should be the on-time delivery of delightful customer experiences that meet a high quality bar. That's what managers should reward you for during production.

The result of prototyping should be insight and innovation that changes how you think about your product, service, and customer. That's what managers should reward you for during prototyping.

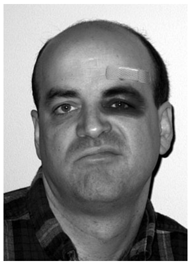
Do yourself a favor

The bottom line is that if you write only one prototype for your idea, you've done your idea, your team, your company, and yourself a disservice. You won't discover all the implications of your idea, good and bad. You won't discover new ways to implement your idea or expand upon it. You won't discover all the uses for your idea or how it connects with other ideas. All you'll have is your first guess. You'll have cheated yourself and your idea.

Remember, it doesn't take longer to implement dozens of prototypes—it takes a different mindset. Sure, you don't want that mindset when you're producing shipping products and services. But during the planning and experimentation phase, timely torrents of tossed-together trials are just what you need to "explore the space." Who knows? Maybe you'll make a few gold records of your own.

Eric Aside One last point about prototypes that a friend of mine in UX mentioned. Some prototypes are "concept" prototypes. Concept prototypes are special. They are blue-sky, over-the-rainbow prototypes meant to see what's out there on a conceptual level. They aren't going to be a real product or service anytime soon. Their purpose is to see where truly broad thinking can take you, which might inform the more practical path you have for the next version or two.

February 1, 2009: "Green fields are full of maggots"



As I said in "Nailing the nominals" (in Chapter 5), the two keys to successful big projects (100+ KLOC) are thinking ahead and defining done. Thinking ahead is about design and planning. Defining done is about setting a quality bar and sticking to it. Yet many big projects go astray even when people think ahead and define done. Why?

Often failure is due to poor executive decisions that place their own agendas above shipping. (Given you hit your quality bar, shipping is better—much better.) However, an even more frequent form of failure comes from engineering teams over thinking and over generalizing—trying to solve world hunger instead of feeding the kids in front of you.

Eric Aside Once you've hit your quality bar, why is shipping always better than messing around with the code? After all, there may be competitive features an executive really wants added at the last minute. How bad can that be? Oh, it can be really bad.

Before you ship, everything is a guess—postponed bug fixes, key features, and design decisions are all guesses until you ship. Once you ship you learn the truth—now you can iterate and improve. Putting off the truth in favor of more guessing is never profitable. It upsets customers and partners, and the continuing chaos corrupts the code.

It's so tempting, so intellectually pleasing, and so self-serving to examine the customer problem you're solving and see the bigger problem, the more general problem, the problem you can solve for all time, all peoples, and all nations. Oh spare me, and spare your customers. Cultivating the green fields of broad ideas is not only self-serving, it's a recipe for feature-rich and value-poor products and services your customers will use begrudgingly, despise utterly, and abandon gleefully at the first opportunity. Green fields are full of maggots.

The horror

Yet, I can't walk down a hallway without hearing people trying to come up with ideas, algorithms, or class structures that "will work for this problem, and all problems like it." Evil. Evil! Warning! Watch out. This line of thinking is EVIL!

What's so evil about general solutions? After all, your code could be both a floor wax and a dessert topping. There are three primary pitfalls:

- You rarely work through the full general solution in one ship cycle. The unfinished framework isn't quite right, but you've shipped it. Now you are stuck with an unworkable legacy code base—forever.
- You introduce a broader test surface and a broader security attack surface. Neither is desirable.

- You put the problem at the center instead of the customer. When the customer isn't at the center, your code loses its soul. It goes from being astounding to being adequate, from marvelous to mediocre.

Eric Aside Why is the unfinished general framework not quite right? Because it's impossible to anticipate everything you and your customer need. After all, you are foolishly solving a general problem, instead of wisely solving a specific problem that you have a chance to iterate toward and get right.

You saved me from this fanatic

Before we break down the three primary pitfalls of general solutions, I need to calm down my Agile-zealot readers. Because Agile methods put a premium on working software and customer collaboration, they tend to avoid the over-thinking trap. In particular, Test-Driven Development (TDD) and emergent design deliberately focus on keeping solutions as simple as possible and laser-targeted at customer requirements.

Because Agile methods avoid the pitfalls of general solutions and general thinking, many Agile zealots believe all big design up front is vile. They are wrong. Sure, the regular refactoring and rework needed for emergent design isn't a problem for small code bases. However, when you get above 100,000 lines of code serious rework becomes extraordinarily expensive. That's why customer-focused, up-front architecture is essential for large projects.

Eric Aside This was researched by Dr. Barry Boehm et al. in May 2004. They found that rework costs more than up-front design on projects larger than 100 KLOC. The larger the project, the more up-front design saves you.

The good news is that many Agile methods, like Scrum, continuous integration, and Test-Driven Development work well within a large project with a customer-focused architecture. These great techniques can keep a team locked onto the customer instead of straying off into the green fields of self-gratifying intellectual exercises in futility.

Eric Aside TDD is an eXtreme programming (XP) and Agile technique for implementation design that produces tight, robust code. As a side benefit, it provides unit tests with great code coverage. The process is fairly simple:

1. Write a new unit test for a requirement of the API or class.
2. Compile and build your program, and then run the unit test and ensure that it fails.
3. Write just enough code to make the new unit test pass (and the old ones).
4. Refactor the code as needed to remove duplication.
5. Repeat until all API or class requirements are tested.

Who will save your soul?

Okay, back to the three primary pitfalls of general solutions— incomplete, unworkable legacy code; expanded test and attack surface; and losing your soul.

Say you are developing a big product like Halo or SharePoint. In either case, you can start with a general framework or you can start with a customer story.

The green fields approach is to start with a general framework. The framework supports all forms of weapons, landscapes, and content viewers. Putting together a game or a website is simply a matter of choosing the desired combination of weapons, landscapes, and content viewers. Bingo! You've got yourself a cruddy game or website.

The game or website has no soul because you focused on the framework instead of staying with the story. In addition, to test the framework you've got to consider every combination of weapon, landscape, and content viewer—each of which also presents an attack vector for hackers. And, should you be foolish enough to expose the framework as an "extensible interface," all these problems become orders of magnitude more diabolical.

Eric Aside You should still have frameworks to help organize your code and classes, but the frameworks shouldn't be general purpose. They should serve a particular purpose defined by the customer story.

Anyone who has played Halo or compared recent versions of SharePoint to the original version knows the value of focusing on the customer story.

It's not gonna get any easier

What's even worse is when you design and develop Halo II and SharePoint 2.0. Inevitably, there are many considerations you didn't address in the prior release that invalidate significant portions of your framework. Unfortunately, those portions are already built and shipped, so you get all the enjoyment of legacy backward compatibility with little of the benefit. Aren't you glad you were just so very clever to focus on the general problem instead of the customer and the story?

"But frameworks and extensible interfaces are critical to our platform!" scream the clueless cretins. Yes, they are when the customer is a developer and the story involves building on our platform. However, the customers of Halo and SharePoint aren't primarily developers. They are consumers and enterprise customers. The story line for them involves crushing the Covenant and sharing information. Focus on the customer, not on the framework.

Can I tell you a story?

What does it mean to focus on the customer and the story? It means don’t solve the general problem—solve the customer problem. The problem customers are trying to solve.

It means knowing the story line (the scenario) for customers. Who are they? What are they trying to accomplish? How are they used to accomplishing it? How might our software help? What would that look like to the customer?

Remember, many of our customers are developers. It’s no different for them. Who are they? What are they trying to accomplish? How are developers used to accomplishing it? How might our software help? What would that look like to the developer using our platform and tools?

Once you are focused on the customer and the story, design, develop, test, and deploy that story for those customers—nothing more, nothing less. Make that story compelling and delightful.

Along the way, certain generalizations will emerge. Avoid them unless they are needed for the story. Only engineer what’s needed to realize the story for the customer. Everything more general is wasted effort because by the time you need it the story will be different.

Eric Aside You are better off following the YAGNI philosophy of software development—only implement what you need when you need it. Never implement what you might need for a later feature. YAGNI stands for “You Ain’t Gonna Need It.” Sometimes the truth isn’t polite.

Temptations always come along

So you code up one story (scenario) and start working on the next. Perhaps it’s even the next release. Aren’t you going to wish you had generalized? No. No, you aren’t. If you had generalized you would have done it wrong. You wouldn’t have known what you know now.

Say that while working on the next story, you realize you need to add a new weapon, landscape, or content viewer. No problem—refactor the design and add what you need now that you know what it is and why you need it. The result is better code, tighter code, and better-tested code because the tests can focus on the story line.

Focusing on the customer and their story line isn’t difficult conceptually or even in practice, but it does require restraint. It is so tempting to solve the big problem for humanity. Resist that temptation. You can’t please everyone, so do your best to at least please your customers. In time, humanity will sing your praises.

Chapter 7

Adventures in Career Development

In this chapter:

Most of the engineers I know and admire are not overly ambitious. They'd be perfectly happy with the simplest of titles and little fanfare so long as they were given a chance to put their ideas into action and have a positive impact on the world. It's true, I kid you not.

Unfortunately, even the most humble of engineers must actively manage their careers or risk losing opportunities to have the positive impact they seek. It's not because the world is cold and heartless, though it sometimes seems that way. It's because the world is highly populated with talented people who all wish to be heard. It's naïve and foolish to leave your chance at destiny to chance.

In this chapter, I. M. Wright shares the secrets behind developing a happy, successful, and satisfying career. The first column tells managers that ambition does not correlate to value. The second one uncovers what it takes to excel in a competitive world. The third column explains the differences between roles and career aspirations. The fourth describes the importance of personal networking to your effectiveness and goals. The fifth covers how to find a new role. The sixth is about developing strategic insight. The seventh reveals how to spot and exploit opportunities. The eighth describes how to avoid writing bogus commitments. The ninth gives you step-by-step instructions for getting up to speed on a new team. The tenth moves you through all the career levels. The eleventh exposes *the tradeoffs of being an executive. The twelfth guides you to becoming a senior individual contributor.*

The most basic career question engineers face is, "Must I be a manager to be successful?" Simply put, the answer is, "No." If your manager tells you otherwise, he is wrong, ignore him. Better yet, switch groups and work for a capable manager. If your boss suggests you try management, consider her suggestion. Management can be rewarding and fulfilling, and your boss may see unmet potential for you in that area. However, it's a choice, not a requirement. The employee data backs me up, as does my own experience and the experience of my friends. You own your career: you decide, you drive, and you draw your goals within reach.

—Eric

December 1, 2001: "When the journey is the destination"

Do we all have to be superheroes? Is that the Microsoft way? When a dev busts his or her behind for you day in and day out for a year shipping your product, is the message always, "That isn't good enough"? What does it take to satisfy the life-sucking career advancement beast?

There are plenty of folks, myself included, who believe and pontificate that you can always do more, be better, and reach higher. Never be satisfied with the status quo, we say; always push yourself and your team to the next level. We associate this attitude with Microsoft Competencies like self-development, drive for results, and technical passion and drive. Does this mean that if you're happy with where you are and what you're doing that you are a parasite on the corporate host?

A man's got to know his limitations

If you think great but satisfied people are parasites, you are a fool. Not everyone can or even wants to be Bill Gates, much as I like the guy. We all have our limits and priorities, and for many of us, being a VP just isn't realistic or desirable.

Ah, but I know what you alarmists are thinking: Lowering your sights means lowering your expectations. Lowering your expectations leads to mediocrity. And mediocrity is a noxious lesion that grows, infects, and consumes a company, rendering it a mountainous boil of pus. So true.

And yet, it can't be right that great devs with years of shipping experience, plenty of passion for product, and drive for results should be told that they have no place at the company unless they become an architect or manager. Surely we've all known folks like this. They love working here. They love coding. They love the software industry, the new technology, and the dramatic positive impact we all get to have on our society.

There is nothing wrong with the attitude and ambition of these devs. They are the backbone of the company. They are the very foundation that drives us forward and makes us successful. I call these dedicated people "journey developers" (because "journeyman" is not PC, and as you know, I am so PC).

Eric Aside "PC" in this case means *politically correct*. By the way, this column received one of the most universally positive responses of any I've written.

Vesting but not resting

Don't get the journey developers confused with the "rest and vesters." The rest and vesters have lost their passion for product, their commitment to customers, and their drive for results. The journey developers still have all these qualities and are totally committed to shipping the best software on the planet, while leaving the people, technology, and design leadership to someone else. Maybe someday they might change their minds and decide to lead by more than just their example and experience. But until then, it is essential that their productive, dedicated, and invaluable contributions be supported, rewarded, and encouraged.

Eric Aside "Rest and vest" was an internal term for employees who stopped working hard and were just hanging around waiting for their stock options to vest. After the Internet bubble burst, the term didn't really apply anymore.

I wish they would only take me as I am

Being a journey developer needs to be a recognized growth path for devs, just as the expert/architect and lead/manager paths already are. It's true that by choosing to limit their leadership, journey developers necessarily limit their impact and influence. This means that journey developers will rarely be able to rise above a level 63 SDE.

Eric Aside In the United States, the entry level for a software development engineer (SDE) is level 59 (compensation levels vary by region). Level 63 is considered a "senior" level for engineers. Levels beyond that take you to "principal" and "partner."

Some might argue that this is reason enough to discourage this career path, but that attitude shortchanges us all. We need their contribution and experience. We can't afford to hand over many of our best developers to some lesser company. Our compensation package is generous enough to keep our journey developers. We can easily find creative ways to reward their efforts, and at the same time not force responsibilities beyond their level.

Of course, some journey developers will change the path they're on and become our future leaders. For them, and for new developers who want the flexibility, the journey developer career path would provide the time and respect that they need to grow and contribute on their own terms. Being patient and understanding as a company can make all the difference in retaining great contributors and increasing their job satisfaction.

We're in this together

As you might have already surmised, the journey path doesn't just apply to development. There are similar people in all disciplines of work at Microsoft who love their jobs and do them extremely well, but they aren't either ready or willing to take on a leadership role. We can and should understand and appreciate those people who love the company, love their jobs, love our customers, and want to contribute without being forced to lead.

As managers, it's important to push our people beyond their comfort zone at times to get them to exceed their own expectations; we can't and shouldn't lower the bar for anyone. However, that doesn't excuse bullying some of our best people into a job they don't want and making them feel unappreciated or inadequate at the very time they are helping us succeed. Being the world's smartest, most productive, and passionate people working for the best company the earth has ever seen should be good enough.

Eric Aside The concept of journey developers was implemented a few years ago. Now no questions are asked when a productive senior engineer chooses to go no higher.

October 1, 2002: "Life isn't fair—The review curve"

Two devs, Harley and Charlie, same level, same division. Harley works hard and well; Charlie works hard and well. Harley is smart; Charlie is smart. Harley is a team player; Charlie is a team player. Harley works on a high-profile feature; Charlie works on a low-profile feature. Guess who comes out on top in calibration? Guess who gets the 4.0 instead of the 3.5? You know it's Harley.

Eric Aside The numbers refer to the old Microsoft rating system, which ranged from 2.5 to 4.5 (the higher the rating, the better the rewards). While a 3.0 was acceptable, most people pursued and received a 3.5 or higher. We've changed the review process, like the rating system, many times at Microsoft, but it's always been about comparing and calibrating your work to the work of others doing the same job at the same level of responsibility.

I can't tell you how many times a Charlie has walked into my office or the office of another manager to whine about how he got shafted. "I worked just as hard as Harley," says Charlie. "You assigned me to the crud feature. I followed what you said and I got the frigging staff. It's not fair." Sorry, Charlie—life isn't fair.

Hear that crashing sound? It's the Charlies of the world leaping out of their seats and screaming, "What the heck do you mean, 'Life isn't fair'? What kind of excuse is that for your poor management! You delegated this crud to me. Am I supposed to blow off your assignment and then drive a truck over Harley and steal his? Who GPF'd your brain, and why am I still working for a manager with a blue screen for brains?"

Welcome to the real world, Charlie. It's harsh and switching managers may not help.

Eric Aside GPF means general protection fault, which crashes old computer systems, causing them to display a blue screen with crash information. As for Charlie's system crash, from his perspective he was set up for failure, when in fact, as I describe next, Charlie had choice and opportunity but passively allowed himself to fail.

I'm not going to take this anymore

Sure, management matters (to borrow a phrase), but it's your career and managers have to juggle many different demands. Someone has to create the less glamorous features, and if you are willing to do it, well you've made your own bed. I hope it's comfortable.

But no one says you have to lie down and take it. Wake up, grow up, be your own Harley—hit the throttle and put your career in the passing lane. "How?" you say. Get wise or eat Harley's dust.

The Harleys of the world are either lucky, savvy, or both. Because we've already concluded that your luck stinks, you'd better get savvy. Here's the key: the savvy dev knows the customer and knows the business.

Eric Aside The names "Harley" and "Charlie" were originally "Hughie" and "Dewey." (I enjoy "Car Talk.") When I came to the critical "Sorry, Charlie" line, I changed names. The only neutral name I could rhyme with Charlie was Harley, as in "Harley Davidson." That set the tone and metaphors for the rest of the column.

Knowledge is power

Sometimes high-profile features are obvious; everyone can point to them from the beginning of the product cycle. However, there are always features that don't look high profile but will be, and some seemingly high-profile features that turn out not to be. To discern the difference, you, Charlie, must understand your customers and understand your business. Typically, most of your team, including management, doesn't.

Why don't many of your peers and managers understand your customers and business?

- Because they think they understand, but are relying on old information.
- Because they think the customer is just like them and the business is to "make lots of money."
- Because they are relying on someone else to understand these things.

You don't need to be so foolish and ignorant.

Taking care of business

To align your work with the business, you've got to research your group's business model. Remember, your boss may not be any more familiar with it than you are. Chances are good that your product unit manager (PUM) knows it fairly well and that your PUM's boss knows it quite well. It's easier to get an appointment with your PUM, so start there.

However, a chat with your PUM may not be enough. A PUM's view of the business model may well be skewed to overemphasize her own products. To really understand the bigger picture, you often need to go to the director, general manager (GM), or VP. As an important courtesy, seek approval from your PUM first, then set up a meeting with her boss.

Uh oh, it's Charlie again: "Why in the world would my director/GM/VP ever want to see me? Don't they have better things to do with their time?" Assuming that they have a free 30 minutes, most of these high-level folks would actually love to see you.

Eric Aside The days of PUMs are nearly over at Microsoft, though we still have directors and VPs. You can read more about Microsoft's switch away from product units in my Chapter 10 column "Are we functional?"

Go ahead, make my day

Sure, a director/GM/VP's life is full; they deal with politics, defend their division and strategy in meetings, put out fires left and right, scrub data to support their positions or find weaknesses, and all sorts of other tasks completely abstracted from the real work that you do—the real work that they did when they started out. Trust me, they miss it.

These high-level folks love to talk to real people. It helps them get in touch with what's happening on the front lines. They love to talk about the business with folks who'll admire them and politely ask them questions that are easily answered. It's a little slice of heavenly pie for these VIPs, and you are the essential ingredient.

One thing to remember is that high-level people deal with problems all day. They don’t want to hear you complain; if you use this time for whining, they will associate you with your problems. When you ask for the appointment, make it clear that you want to talk about where you and your team fit into the business.

Eric Aside As an employee, you have every right to complain to upper management about any problem you see, any time you want. You shouldn’t keep problems to yourself—you should seek solutions. That said, this particular appointment with your director/GM/VP is about understanding the business and getting to know your leadership on a personal level.

Reach out and touch someone

So, you’ve talked to your upper management and charmed them while you learned about your business model. (This will also come in handy at promotion time.) Next you need to understand your customers.

Charlie says, “Isn’t that the PM’s job?” Sure it is; just don’t choke on Harley’s exhaust as he rides off with your rating. It is everyone’s job to understand the customer, and there are plenty of easy ways to do it:

- Observe usability tests.
- Attend customer visits and focus groups.
- Join an online community.
- Talk to the Product Support Services representative for your products.
- Read reviews of your products.
- Check out reviews of your competitor’s products.

Do whatever it takes to get in touch with your customers. Know what they want and what they really care about.

Got lemons? Make lemonade

Now you’re ready to slap on some chaps and fire up the hog. If you know the business and know the customer, you can tell which features are juicy and which are window dressing. You’ll also know which features aren’t getting the attention they need to really win the customers’ business.

Eric Aside “Hog” is an affectionate nickname for a Harley Davidson motorcycle.

Chances are you already work on some not-so-flashy features. Recognize that they have the potential to be key to your product's success. They just need to be focused on serving your business and customer needs. Move them in that direction and you'll become the team hero.

Change your tune

If you find that others have all the sweet assignments, or you just can't stand to work on backward compatibility, legacy support, and setup (which are key but not exciting) anymore, there are other ways to change your fate.

The easiest way is to choose the best features to work on at the beginning of a product cycle. Because most folks are ignorant of business and customers, you shouldn't find it difficult to identify some prime features that others will miss. However, if you're already in the game and feeling shut out, there are two sure ways to get yourself back in:

- Look for late-breaking feature needs. These are the "whoops we forgot that" feature, the "last-minute customer request" feature, the "keep up with our competitors" feature, the "new cool enabling technology" feature, and the ever popular "this critical feature stinks—we need new blood on it" feature. Remember your business and customers to avoid jumping onto a loser; but if it's a winner, get all over it. If the boring, stupid features that you were working on suffer as a result of your new assignments, they should have been cut anyway.
- Become the backup dev for a great feature. Go to your boss and say, "Hey, Harley is working on a pretty critical feature without any apparent backup. If it's alright with you, maybe I could learn more about it, maybe help him fix bugs or something, and then we'd have a backup in an emergency." Bingo, you're in. Either you get to be the hero this go round or you get Harley's feature next go round when he moves onto something else.

Eric Aside I talk more about taking advantage of opportunities later in this chapter in my column "Opportunity in a gorilla suit."

The one behind the wheel

Life really isn't fair, and the good assignments don't just get handed to you. It's true that your boss should understand your business and customers, and share the wealth in assignments. But you can't count on your boss's knowledge or generosity to move your career forward.

Take ownership of your career and the work that you do. Know your business and customers well. Apply that knowledge to make your products better and drive your career forward. If you succeed, it won't just be you who wins, we'll all reap the benefits.

Eric Aside There are three columns that hold a special place in my heart: "Dev schedules, flying pigs, and other fantasies" (my first column, which appears in Chapter 1), "Where's the beef? Why we need quality" (a turning point on the road to quality, which appears in Chapter 5), and this column (critical topic and a fun read). Many great topics, rants, and word plays followed, but these early columns set the tone for the rest.

November 1, 2006: "Roles on the career stage"

I'm sitting here and I'm tired. It's not long hours. It's not a cold coming on. It's not a lack of sleep. It's the idea of explaining for the thousandth time how someone with three reports can be an individual contributor or why there isn't a special Career Stage Profile for an architect. The answer to both questions is simple: role ≠ career stage. Get it? I didn't think so. Darn, I'm tired.

Eric Aside Career Stage Profiles are detailed descriptions of the work expected of employees at different career stages for different disciplines. Generally, for each discipline there are three sets of stages: entry stages (for U.S. developers, that's levels 59–62), technical leadership stages for individual contributors (levels 63–69), and organizational leadership stages for leads and managers (levels 63–69).

Look, you're not dumb and neither are the other thousand people I've temporarily enlightened. There's some subtlety to this; words get overloaded and confused. But it really isn't complicated—role and career stage just aren't the same thing.

One, in time, plays many parts

Perhaps some definitions will help. A role is like a part in a movie. Different people can play the same role. The same people can play more than one role. In fact, you can play multiple roles in the same meeting or conversation. It happens all the time.

For example, say you are a dev lead discussing a feature implementation with an old friend who happens to report to you and is also married to your cousin. In the same conversation, you could play the role of manager, friend, architect, relative, peer, mentor, and adversary. That's an extreme case, but perfectly plausible.

Roles can change by the minute, but typically at work we have a primary role, perhaps a secondary role, and a handful of lesser roles. For instance, some architects have a small number of reports. Their primary role is architect. Their secondary role is lead. Their lesser work-related roles include Microsoft employee, peer, tester, designer, mentor, and subordinate.

Stage right

Career stage is quite different from role. Career stage doesn't change by the minute. You don't have multiple career stages—you've got one. You might reference a number of career stage profiles in your commitments because they help define the roles you play. However, you are still only residing in one stage at any given point in your career. (I'll explain why later.)

So what is a career stage? A career stage defines your progress in your chosen growth path. Because you are only in one career stage, knowing which one is important. To validate your current career stage follow these steps:

- Select your discipline: dev, test, PM, UX, and so on.

> **Eric Aside** UX stands for the *User Experience* discipline, mostly designers and usability experts.

- Select your career aspiration: technical leader (individual contributor stages) or organizational leader (manager stages).
- Review the career stage profiles with your manager to fit your skill set, discipline, and aspiration to a career stage. (Your current level and region often provide a clear starting point.)

I aspire, sir

Remember, it's not your current role that matters. It's your aspiration. Say you are in a dev lead role, but you want to be known for your technical leadership, not your management skills. If you are a developer at level 64 and work in Redmond, theoretically your career stage could be either Lead SDE or Senior SDE. However, because your aspiration is to be a technical leader your career stage should be Senior SDE.

"But I'm a dev lead," you say. Yeah, if you want to stay a dev lead or perhaps one day become a dev manager, that's fine. But if you want to become a technical leader, then either switch your career stage to Senior SDE or enjoy being dead-ended as a Lead SDE.

"But I have three reports," you say. Yeah, so one of your roles is a lead. Big deal. That has little to do with your chosen career growth. If you want to grow as a technical leader, then you must choose that growth path. Sure, you still need to be a good manager in your lead role and many of your commitments will still come from the Lead SDE stage, but those won't be your career development goals.

Overqualified

"But what if I'm a level 66 Principal SDE and I'm happy being a lead?" you say. Lead SDE tops out at level 64 in Redmond (level 65 is in the accepted range). At level 66, you are overqualified to be in the Lead SDE stage (though you certainly could still play the lead role).

Why does the Lead SDE stage top out at level 64? Because at Redmond level 64 you should have all the skills and scope the lead role requires. If you are level 66 or higher, you didn't get there because you could manage a small set of individual contributors really well. You got there because of your technical leadership. If you stopped being a technical leader and played only the lead role, you'd be underperforming as a level 66. That's why your career path is technical leader and your career stage is Principal SDE.

Eric Aside Leads topping out at level 64 was the biggest point of contention about this column. People, important people I respect, thought I was saying that dev leads can't be above level 64. They misunderstood. My point is that if you are above level 64, you must have more going for you than managing a small team. It must be an important team, and you must be providing strong technical leadership. Thus, your continued growth isn't the result of your small-team management skills, it's the result of your technical insight and guidance. You want your career stage designation to match your source of growth, while maintaining your secondary role as dev lead.

I'm special

"So why is there no architect career stage?" you say. Being an architect is a role. You need different career stages for different growth paths, but not for different roles. The growth path for a highly technical specialist is the same as the growth path for an architect. Both are technical leaders, so they both share the same career stages.

Sure, the role of an architect is different from that of a world-renowned expert in database logging. But they must have similar skills and develop in similar ways to advance. Thus, they are both either Principal or Partner SDEs.

There can be only one

"Okay, fine, but why can you be in only one career stage?" you say. If you can play multiple roles, why not have multiple career stages? Because your career stage is tied to your growth path, which is tied to your aspiration.

Most healthy individuals with single personalities have one career aspiration at a time. Yes, it can change from time to time. But at any given point in your life, you have one primary career goal. That one goal determines your career stage uniquely.

Now you could conceivably aspire to be a major technical leader and organizational leader simultaneously, but you'd fail in a miserable way. The reason isn't due to a lack of effort. Enough amphetamines could keep you working 24 hours a day for weeks at a time.

The problem is that being a leader means leading others. That means communicating with them, but not everyone is around all the time. At the end of the day, the availability of others constrains how much leading you can do in a day. To be a great leader, you must focus your efforts.

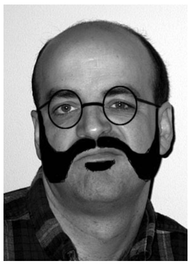
What do you want to be?

What do you want to be when you grow up? That's the central question the career model asks. You don't need to have the perfect answer. You just need an answer for now that can focus your efforts and allow you to make progress.

"Pick one and go with it" is the advice I prefer. You can switch later if your aspirations change, but meanwhile you'll learn and grow. We don't always become what we set out to be. It's the journey that holds the interest. But you can't just stand undecided at the fork in the road. Choose your path and enjoy the ride.

Eric Aside It's been nearly five years since I wrote this column, and people still confuse roles and stages. To avoid confusion, Microsoft has considered adding a lead path to the engineering disciplines alongside the IC and manager paths. Personally, I'd drop the IC and manager paths altogether. I'd separate being a lead or manager as a role. All developers would be developers, but some would have management roles, architect roles, web service roles, and so on.

May 1, 2007: "Get yourself connected"



For many Microsoft engineers, it's a matter of principal. Not *principle*, as in belief, but *principal* as in that coveted career stage when they get more influence, more esteem, and more stock—as if one begot the others. That's ridiculous and naïve. In fact, it's influence and esteem that help you reach the principal stage, with the extra stock as a nice side benefit.

But how do the *unprincipaled* acquire influence? How do they receive esteem? Many engineers feel they are entitled to these gifts because they are smart and uniquely skilled. Okay, their capability and intelligence may be unique among their relatives, but not among their peers. We expect everyone we hire to be smart and skilled. So where does that leave you?

Well, you could become an organizational leader, like a lead or manager. Do those jobs well and influence and esteem are yours from those you manage. But management has its own issues and isn't for everyone. How do you broaden your influence as an individual technical leader when being smart isn't enough? Through your network—networking is fundamental.

It's who you know

The cynical among you might chafe at the idea that who you know is important. After all, aren't your ideas your true value? Yes, of course they are. Now, who are you again? What's your idea? Why does it deserve 20 minutes of my time to understand it, relative to the ideas of the other 27,000 smart engineers at Microsoft, let alone the innumerable smart engineers across the world?

It takes time to comprehend and appreciate ideas, especially from people you don't know because you are missing all their context. Therefore, if you want your ideas to be appreciated, you need to develop relationships with people who can appreciate them. The more people you know, the more people there are who can appreciate your ideas, the more influence you have, the more esteem you receive, the further you will go. Simple.

A strong network can also help you find new roles, get better reviews from peer feedback, and learn more about everything that is happening. Your network makes you look, act, and be smarter. When someone asks you a question, you don't need to know the answer; you just need to forward the question to a smart friend. Even though the smart friend had the answer, the questioner still sees you as the person who got the answer.

But how do you expand the network of people you know? And once you do, how do you keep all those relationships active? Let's face it, most engineers, myself included, are not social butterflies. We also don't have the time or inclination to host dinner parties. Luckily, networking isn't that difficult or imposing. It does involve a time commitment, but not as much as sales or politics.

I use habit and routine

To build a large and strong network, you must acquire certain habits:

- **Be curious** all the time about everything.
- **Be appreciative** of those who help you.
- **Be responsive** to those who ask you for help.

These three habits are essential to building and maintaining a strong network. They aren't complicated. They don't take as much time as personal hygiene. But if you let them slip, your network will disintegrate quickly. So when I say "make them a habit," what I mean is, *make them a habit*.

Aren't you curious?

Most engineers are naturally inquisitive. The trick to building a network is to apply that curiosity to other people's interests. After all, who isn't charmed by those who show genuine interest in what they do?

When you bump into someone you know, or even someone you don't, ask about their work. Find out as much as they are willing to tell within the time available. Make it a habit to do this all the time. You will learn a great deal, and you'll develop a broad network.

It's important to focus on the other person, not yourself. This isn't about you; it's about your acquaintance. Of course, you should answer any questions your new or old friend has about you, but don't stray far from your friend's interests. After all, talking about yourself gives you little value other than feeding your ego. Talking about your friend develops familiarity, and with it, trust. You'll get to focus on your interests when you have the need later.

> **Eric Aside** A question I received about this column was, "What if the other person is trying to build a network—doesn't one of you need to talk about personal interests?" Yes, of course, the conversation should be balanced. I emphasized not talking about yourself because it's so easy to get lost in your own interests and exploits.

So, next time you're stuck on a bus to some event, stuck in line for food, or stuck waiting for a meeting or class to start, ask the person you're stuck with about their current project. Not just the name of the project, but what it is, how it works, what's tricky about it, what's fun, what's a pain, what are the people like, what is the management like, everything! This isn't small talk—this is genuine curiosity. It's just what you need to engage the person into a mutually beneficial relationship. It works and costs no more than the time you were stuck anyway.

You have our gratitude

Another subtle yet effective way to draw people into your network is to owe them a favor. I learned this reading about Abraham Lincoln, who borrowed books from neighbors who often lived miles away. In addition to getting access to books that were scarce, Lincoln found that asking for a small favor, such as borrowing a book, created a strong bond between him and the lender. The lender would find Lincoln to be trustworthy and appreciative. In addition, Lincoln would be in the lender's debt—a situation advantageous to the lender and one that made a continuing relationship desirable.

So, say you are looking for an opportunity to draw a specialist in media codecs into your network. You know the person's address, but they don't know you. Graciously asking that person for help with a codec issue would be an ideal way to start a relationship, assuming you follow certain guidelines:

- Use a real issue that requires the help of a specialist. Anything less will waste the specialist’s time and be met with disdain.
- Be clear and concise in your question, and provide a time frame. Again, this demonstrates that you value the specialist’s time. As I’ve written before, this is important in all communication. (See “You talking to me? Basic communication” in Chapter 8.)
- Thank the specialist generously; detail the value you received from their help. You want the entire experience to be positive for the person helping you, making them feel valued for their expertise.

You might say, “But why not do the specialist a favor? Isn’t asking the other person for a favor completely twisted?” When you do a favor for someone else, particularly unasked, that person now owes you. The relationship with you is tainted with guilt and associated with burden. But when the specialist does you a favor that you truly appreciate, there is no burden or guilt. Instead, your new friend only experiences feelings of advantage and being valued.

I’ll get back to you

Naturally, favors work both ways. That’s how networks function. People provide service to you. You provide service to them. If you want to keep these relationships working, you must be responsive.

What does “responsive” mean? Well, how long does it take you to think someone doesn’t care enough to respond to your mail? One day, perhaps two? You must respond to mail from your network within roughly 24 hours. Period.

Your response could be, “I’m sorry, I can’t get you an answer right now. Can I get back to you in a week?” That response is far better than none at all. No answer means you don’t care, whereas the “I’m busy right now” answer means you care, but you’re busy.

You might say, “You’ve got to be kidding! I’ve got too much e-mail as it is. Now, I’ve got to answer every stupid question that comes to me within a day?” First of all, maintaining a network is an investment that pays large dividends. Nothing comes for free. Second, answers often come cheap. Forwarding a question onto someone else is just as good as answering it yourself. Why? Because people only care about getting an answer. They write to you and they get answers; that’s all that matters. Just remember to appreciate those who answer the questions for you.

Eric Aside I provide many tricks to managing your mail efficiently in “Time enough” in Chapter 8.

Welcome to the world

Where do you find people to be in your network? Everywhere. In lines, meetings, and classes; on discussion aliases; in collaboration spaces like CodeBox and Toolbox; from web queries; and everywhere else you go at Microsoft. Outside Microsoft, you can find people at conferences, on blogs and forums, you name it. Finding people is easy. Getting them into your network and keeping them is the trick.

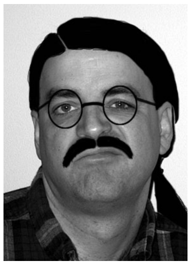
When you find people, keep track of them. I keep great e-mails and papers that I read, so I can contact the author(s) at a later date, when there is an opportunity to draw them into my network.

> **Eric Aside** There are a bunch of personal networking sites now on the web. While they are very clever and useful, be aware that Brad Pitt isn't likely to make you a personal friend, and neither is the Angelina Jolie of your field. These sites are a great place to get started but not the complete solution.

Once you get someone into your network, stay in touch. There's no need to be artificial about it. Building and maintaining networks is highly opportunistic. The key is to take advantage of opportunities. When you bump into someone you haven't heard from in a while, stop and talk even if it's just for a few minutes. Ask them about what they are doing now. Be curious. Draw them back in.

Remember, networking doesn't have to be a great deal of work, but it is a commitment. The payback is expanding your mind and your reach. You will learn more about what's going on and where opportunities are. More people will know and respect you and your talents. Your ideas will be more easily accepted. All that leads to greater influence and esteem. You'll even make some close friends along the way. That's a *principal* worth pursuing.

September 1, 2007: "Get a job—Finding new roles"



It's the end of review season: time to reflect on your career and current situation. Some people have a career plan, know where they're at, and already have their next move lined up. I call these people "wise, successful, and yet, disturbing." Perhaps I'm jealous. After all, I should have a multistep career plan in place. But too much life seems to happen to my plans, and I find myself never quite sure of what's coming.

That's no excuse to purely improvise either. Career improvisationists have a name too—"bitter and exploited." You need to plan enough to know what you want and how to get it. Even if you're happy with your current situation, life has a way of changing so it's best to consider your future options.

Of course, you could be ready for a change right now. Perhaps you feel dead-ended, getting by but not making any progress. Perhaps you are driving hard, but your current role is slowing you down. Perhaps you are utterly devastated—your career in shambles; your manager a nemesis; your daily job a study in panicked, misguided, and endless toil leading only to failure, waste, and uninspired mediocrity. It's time to find you a new role.

Eric Aside I think people feel utterly devastated far more often than they actually are. Either way, considering a change would likely help you appreciate your current situation more or help you find a better one.

Now which way do we go?

If the time is right to move on, where do you go? Many jobs open up at the end of reviews because people move on and because hiring managers can't fill positions while reviews are being written (candidates don't like giving their current manager bad news during reviews). This means many jobs will be available. How do you choose?

Naturally, you want a job in which you can be challenged, make a difference, and have fun. However, there are many factors that influence enjoyment, impact, and growth. The common criteria people use to differentiate roles are:

- The technology—what you're doing
- The market—why you're doing it
- The work style—how you're doing it
- The team—who you're doing it for and with

Most engineers choose primarily on the basis of technology. They want to work on something cool; after all, they are engineers. That's a mistake—technology comes last. You probably chose technology first for your current job, didn't you? How's that working out?

We've got a situation here

The most important criteria is the market. Is your new group working on something strategic and important, or is it one VP's bad hair day away from cancelation? There's a joke about a difficult star baseball player on a poor small market team. The manager tells him, "We can lose with you or without you." Don't join that team.

The next most important criteria is the work style. Will the job be a learning and growing experience for you, or is it the same old thing? The career model talks about this in terms of “experiences.” You want variety in order to advance yourself and your career. Big groups, small teams, incubation efforts, different disciplines, and remote sites all provide unique opportunities for growth. Don’t be afraid to seek something different.

However, market and work-style criteria usually only help narrow down the choices. The real differentiator is the team. Whom you are working for and the people you are working with have the biggest impact on whether or not you like your next assignment, by far. The coolest technology can be made maddening by poor management. The most mundane technology can be made fascinating by your peer group. Ignore this advice at your peril.

Eric Aside I give the same advice about college classes. The quality of the instructor is far more important than the subject.

There is nothing here for me now

By now, nearly half of you are probably saying, “That’s all nice, but how do you find jobs that meet your bar? It’s not like managers broadcast their incompetence, and besides, good senior jobs are always gone before you can apply.” Guess what? Hiring managers have the same problem finding good candidates. The solution is the same—cast a wide net and use your network.

Not all potential positions are posted and available, only the currently open ones. Savvy candidates find out about potential positions before they are posted. Naturally, the position still needs to be opened, giving all candidates a chance, but the earlier you know about a job, the better chance you have of getting it. That’s true inside and outside the company.

How do you find groups looking for someone like you, even before they realize it? The same way industry hires land great jobs—they use their network. As I discussed in “Get yourself connected” (earlier in this chapter), having a strong network of peers is essential to your success. When you are looking for a new role, write to all of them. Yes, all of them. Ideally, potential managers will hear about you from three different friends of yours. That will make a strong impression and produce an abundance of leads.

Should you feel guilty if friends get you a great job? That depends. Did you earn it because you impressed them? It’s a competitive market for talent out there. Using your connections is a measure of your influence and impact. It’s the way senior people get jobs, so you better excel at it.

It's been a long time

The next hurdle to your perfect new assignment is the informational and interview loop. Pay close attention to how you are treated and how the team makes its decision. Those indications may signal how the team operates and what treatment you can expect in the future.

Prepare for the informational by learning about the team and their projects. If you've got a strong network, then some of your friends should either be on the team or have worked with them closely. Speak to them first, before talking to the hiring manager, and ask what life is really like on the team.

> **Eric Aside** Informationals at Microsoft are informational meetings you can request with hiring managers before you apply for a position.

As for information about you, be honest and forthcoming. However, no matter how bad your current situation is, never discuss your desire to leave your old team; always focus on your interest in joining the new team. That's critical, read it again. If asked why you're leaving, just say, "The opportunity on your team sounds really intriguing." If pressed you can add, "Sure, my old team is imploding, but that's not the main reason I'm talking to you." (And it shouldn't be the main reason—desperate acts rarely improve your lot.)

Should you be asked back for an interview, you might be nervous. Perhaps it's been a long time since your last loop; they are intimidating. Some people cram like they're back in college. While some mental preparation is quite helpful to boost confidence and refresh your memory, it's not the key to success.

The key is to be yourself. If you try to be someone else and get rejected, you'll never know if they misjudged you. If you try to be someone else and get accepted, you'll worry about living up to false expectations. ("Did they really want me?") Be yourself and you'll be more comfortable and confident, which always helps. You'll also know for certain whether or not you and the team are the right fit for each other.

If I go there will be trouble

What if your current team and manager don't want you to leave? What if you have critical unfinished work? What if those facts fill you with glee because members of your former group are pond scum and deserve every moment of pain and suffering you can deliver? Hold on there a minute, Mr. Kersey. Let's think this through.

Eric Aside I'm referencing *Death Wish*. Yes, I am old.

First of all, no employee is indispensible. Life goes on in catastrophes far worse than you leaving a team. Therefore, guilt is inappropriate for you to feel or for your former manager to put on you. If they can't handle you leaving, then they do deserve what's coming.

That said, this is one of those times when being a professional is critical. Crazy things happen, particularly in our industry. You never know when you might need to work with or even for some of your former team again. It just isn't right to talk badly about your old team or to refuse to provide a smooth transition.

That means being up-front about interviewing, providing a transition plan, and following through within reasonable limits. Keep in mind, we are talking about a transition plan, not indentured servitude. If the work you do for your old team isn't about smoothly shifting your responsibilities to others, then it shouldn't be in the plan.

I must be travelling on now

Change is never easy, which means there's never a good time to do it. However, putting your career on hold is no good for you, your team, or Microsoft. After reviews and after shipping are great times to reflect on your career and decide if a change would do you good.

There's nothing wrong with being happy where you are, but if the moment has arrived for you to go, welcome it. Great managers will support and help you. After all, flow through their team is critical to their long term success (see "Go with the flow—Retention and turnover" in Chapter 9).

Regardless of what's involved, finding the right challenge to learn new skills and become more valuable is well worth the trouble. Microsoft is a big place with a wide variety of opportunities. We hire the best, but you don't stay that way without working at it. The occasional change of scenery may be just what you need to stay ahead, engaged, and energized.

Eric Aside When I started at Microsoft more than 15 years ago, it was still small enough that people had plenty of friends across the company and could easily switch divisions without needing to reestablish their reputation. Now, each division has nearly as many engineers as the entire company did back then. This means that when you switch divisions you must reestablish your standing nearly as much as an external hire. Proceed with caution—if there is a desirable job within your division, you might want to consider it first.

December 1, 2007: “Lead, follow, or get out of the way”

We’re closing in on midyear career discussions again. It’s time to place your hopes and humility in the hands of your hierarchy. I still haven’t recovered from the amputation of our midyear ratings, which allowed managers to send messages and employees to salvage careers after a temporary setback. They’ve been replaced with a time-consuming CareerCompass that contributes complexity and confusion instead of context and clarity.

Eric Aside CareerCompass is an internal web application that allows employees to assess their competencies and skills against standards for their career stage. Managers and selected peers can also assess employees through the tool. It’s a nice idea, but the first version was a less than ideal implementation.

Don’t get me wrong, even though I want the midyear ratings back, I initially found the less confrontational career conversations to be, well…constructive. Unfortunately, though HR and managers may mean well, their advice is often cryptic. Take the biggest offender: strategic insight. The common conundrum: “My boss says I should think more strategically. What on earth does that mean? What should I do?”

Good question. Ask your favorite manager or executive about improving your strategic insight, and he or she will likely suggest getting more involved in planning, focusing on the vision, thinking more about the business, and other such well-meaning nonsense. Yes, strategy happens in those activities, but attending a pro football game doesn’t make you a great quarterback.

Strategy is not an activity; it’s a way of thinking. It happens anytime, not just during planning. If you want to get better at it, wasting more time in meetings isn’t likely to help. You need to mature your way of looking at work and the world. You’re not an ignorant waif anymore. It’s time to wake up and grow up.

Blind Faith or Cowboy Junkies?

A friend was recently comparing two coworkers—a guy who did whatever he liked and a gal who followed the current business strategy. My friend said to me, “[The gal] is a better strategic thinker.” No, she’s not. Neither is thinking strategically at all. The guy is a cowboy without a herd. The gal is tactically contributing to a given strategy, which is far better, but is not strategic insight.

There’s a growth curve for strategic insight. Take a look and see where you fail, I mean, fall:

- **Ignoring strategy** The self-centered cowboy

- **Following strategy** The tactical contributor
- **Questioning strategy** The strategic tactician
- **Evolving strategy** The strategic thinker
- **Determining strategy** The strategic leader

Let's break these down, examine the pitfalls of each level, and discover how you make progress up the curve.

Yippee-ki-yay, project buster!

The self-centered cowboy ignores strategy, considering it bureaucratic mumbo-jumbo that might otherwise prevent him from creating cool innovative products. His heroes are the storied rebels who broke the rules and created the killer features that vaulted them to the principal and partner levels. Oh, to be that bold and bright!

Eric Aside The principal and partner levels are among the most senior at Microsoft. I know a number of these heroes, and none of them ignored strategy. They ignored middle managers who took the strategy too literally instead of truly understanding the intent. The rebels did the right strategic work and were rewarded for it by upper management. The moral of the story is, "Make sure you deeply understand the strategy before you start messing around."

Yeah, right. Never mind what the actual facts might be in those stories; we're not a startup anymore, and most cowboys aren't working on incubation efforts. We are working on established products and services, and cowboys in the big city will get lost and ignored no matter how well they rope or play guitar.

Why do cowboys get lost and ignored on big projects? Because their individual actions are like random noise. Without coordination, each cowboy's efforts cancel out the others. Their features may be cool in isolation, but they don't fit into a larger whole. Thus, no one sees them, no one uses them, and they add no value.

What's worse, all the random features created by cowboys only confuse customers more. They actually do more harm than good. A cowboy, setting out on his own to change the world, is only setting himself up for disappointment. Not because he doesn't have good ideas or because the big city won't give him a chance. A cowboy fails because he doesn't organize with others to drive the product forward.

Resistance is futile

The tactical contributor follows strategy, trusting the plans of the collective and finding creative ways to bring those plans to life. Many cowboys feel that tactical contributors have cowardly relinquished their freedom for the safety of the collective, with a "baaaa" and a "moo" for good measure. However, the real reason for being a tactical contributor is usually one of the following:

- **"I'm not interested in the responsibility of deciding strategy."** As I talked about in "When the journey is the destination" (earlier in this chapter), not everyone wants to become a VP or technical fellow. Some are happy to make their valuable personal contribution and then spend time focused on their other interests. Understanding and following the current strategy is the best way to have significant individual impact.
- **"I'm not prepared to take responsibility for deciding strategy."** For those who aspire to lead technically or organizationally, to expand their impact beyond themselves, they need a place to start. Even if you disagree with the current strategy, your smartest initial move is to learn it and follow it. Only fools try to fight or evolve something they don't understand, repeating past failures. Intelligent people seek first to understand the strengths and weaknesses of the current strategy before they change it.

Following the current strategy doesn't mean giving up original thinking or creativity. It's actually quite the opposite. Many of the world's greatest buildings and works of art were inspired by the constraints of the land or medium in which they were made. Finding ingenious and beautiful designs that meet a fixed set of requirements is a challenge that great minds delight in solving.

Tactical contributors are highly valuable employees, but they aren't quite ready to make the leap to strategic thinker. Before they can determine new strategy, they must learn to constructively question the current strategy.

Is that right?

The strategic tactician questions strategy while still following it. This is not to be confused with idiots who complain about everything without providing any constructive alternative. The strategic tactician is still tactically following the strategy and wants to see it succeed. She simply knows enough and thinks enough about the strategy to question assumptions and approaches that seem suspect.

Some people are afraid to take this step. They don't want to seem like troublemakers, and they trust that decision makers know what's best. For the bashful out there, here are a few facts that might get you to the strategic-tactician level:

- Decision makers are absurdly imperfect, just like the rest of us. They don't always have the right or complete information. They can be blindsided by their own biases. They might even be incompetent, though Microsoft has some real talent in these ranks.

- Troublemakers aren't thoughtful. There's a big difference between thoughtful questions and annoying questions. If you ask a question you could have easily answered yourself, or you complain for the sake of complaining, you are being annoying and causing trouble. However, if you've thought about a situation carefully and truly care about getting things right, you aren't causing trouble. You are pointing out areas that need a better explanation or potentially a change in direction. Most decision makers will thank you for your input and think well of you at the next review. If yours doesn't, perhaps it's time to switch projects.
- The current situation is constantly changing. When the strategy was devised, it was based on what people knew at the time. Today is different. New issues come up. The market, technology, competition, and customers change. Leadership changes. Requirements change, sometimes in subtle ways. All these changes can cause the current strategy to become inadequate or inappropriate. It's your job to notice, check if the strategy still makes sense, and question it if it doesn't.

So how do you notice when to question the strategy? When your gut stops you and asks, "Wait, why are we doing this?" At that point, get the answer. Ask your friends and coworkers "Why?" Keep asking till you get a good explanation or have no recourse but to question the decision makers. They will remember you as a strategic thinker, not a troublemaker. Does that mean you are a strategic thinker? Not quite, but you are ready to make the leap.

I suggest a new strategy

The strategic thinker evolves strategy, adjusting it to account for changing conditions or gaps in approach. The big leap to strategic thinker is that strategic tacticians point out issues, but strategic thinkers resolve them. A strategic tactician brings up a problem with her manager, but a strategic thinker also lays out a practical plan to resolve the problem that fits within the current strategy.

In many ways, it's really not a big jump between strategic tactician and strategic thinker. You just must be willing to take responsibility for change instead of leaving it to someone else.

Often people confuse strategic thinking with breaking the rules or making up your own rules. That's cowboy talk. A strategic thinker evolves the current strategy, coming up with resolutions that stay within its spirit but expand or adjust it. That's why the tactical contributor stage of truly understanding the strategy and applying it is so important.

As a strategic thinker matures, he will start generalizing the strategy, evolving and expanding its impact. For example, say his team is suffering because they aren't documenting their interfaces sufficiently to comply with regulations. It's much faster and easier to document them from the start, especially with tool support, but no one is in the habit.

A freshman strategic thinker might propose adopting Sandcastle (*blogs.msdn.com/b/sandcastle/*) for documenting his team's managed code. A sophomore strategic thinker might add Sandcastle to the product-line build. A junior strategic thinker might add documentation completion statistics to build reports and quality gates. A senior strategic thinker might establish his Sandcastle build and report solutions internally on CodeBox or externally on CodePlex (*www.codeplex.com*) and encourage his network to adapt the solution across Microsoft. A senior strategic thinker is one small step away from being a strategic leader.

I am not a leader of men

The strategic leader determines strategy for her group, her division, or perhaps even for the company. The difference between a strategic thinker and a strategic leader is that the leader defines strategy for whole new areas. While you can become a strategic leader on your own, it helps to have existing leadership sponsor your ideas.

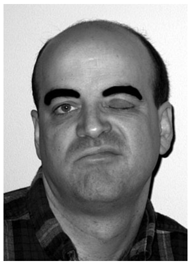
Eric Aside You don't need to be a manager to be a strategic leader. Architects are strategic leaders. Designers are strategic leaders. Experts who work out the next great approach to security, performance, or reliability are strategic leaders. It's an approach and mindset more than a role.

Often this can be done by circulating white papers with new strategic ideas, creating grassroots community around your plans, or taking on leadership roles within your network. However, you must be prepared to lead the effort to bring your strategy to life. When it's your idea, it becomes your responsibility to see it through.

Anyone can become a strategic leader, though not everyone may want that responsibility. The key is to *not* focus on yourself and your interests; they are too narrow and won't engage others to join you. Instead, focus on the interests of your customers, your group, your company, and the world. How can they be better served with the capability and resources we have to apply?

It's so easy to go after your own pet project, but you'd be putting on cowboy boots. Being an inspirational leader means walking in someone else's shoes and serving their needs for a greater purpose. Remember that and you may become a great leader indeed.

July 1, 2008: "Opportunity in a gorilla suit"



It's annual review time at Microsoft. We differentiate pay between high, average, and low performers in the same roles. Thus, it's time to calibrate those who've made the most of their opportunities in the past year with those in the mainstream of solid engineers and those who haven't quite kept pace with peers.

Eric Aside There are many people inside and outside Microsoft who critique differentiated pay, saying it sabotages teams and teamwork. While I do agree team results should be a component of compensation, I don't think differentiated pay is the problem. (See "Beyond comparison—Dysfunctional teams" in Chapter 9.)

As a manager, this is also the time for the whiners and the clueless to lament to me about their lack of opportunities to grow and demonstrate their true worth. As if managers hoard those opportunities, giving them out only in moments of weakness or pity. As if those opportunities are rare—hidden treasures available only to the select few with guile and charm. No, you fools, opportunities aren't rare and they aren't hidden. Opportunities are big, loud, and aromatic. They stand right in front of you in gorilla suits beating their chests all day long.

Yet many smart engineers don't notice. Huge, noisy, smelly gorillas in their face day after day, and they don't notice. Sometimes their manager hands them the opportunity, invites them to a meeting, or puts them on a project, and still the engineers, capable engineers, ignore it. They hand the opportunity to someone else. They give it only passing attention. They leave it sitting in a corner till it finally devolves from inattention.

Why?!? Why don't engineers notice these opportunities? Why do they toss them aside, only to complain in July about the lack of opportunity? Towering, raucous, pungent opportunities in gorilla suits, every day, ignored. Why?

I'm blind, man

I used to think people ignored these opportunities because they were lazy or apathetic. I still believe apathy is a real problem, but I no longer consider laziness a key cause. Instead, I believe people miss opportunities because of a concept called "inattentional blindness." Basically, engineers don't notice flagrant opportunities because their minds are focused elsewhere and aren't paying attention.

There's a telling video you might have seen that asks viewers to count the number of times a basketball is passed amongst a group of people. During the video, a person in a gorilla suit walks into the middle of the group, faces the camera, beats its chest with gusto, and then walks out of frame. At the end of the video, viewers are asked if they saw the gorilla. Not only do people miss it (including me the first time), some people insist the gorilla must have been camouflaged. Then they watch the video again and notice the big gorilla in the middle of the frame, making a mockery of their perception.

It is all around us

Engineers don't notice the opportunities in gorilla suits because they are focused on their day-to-day duties of counting basketball passes. They are too distracted to notice. However, perhaps you are one of those who think the opportunities are camouflaged. Allow me to remove your blinders and list the opportunities that pass you by every day:

- **Killer features** Sure, you know these exist. You might even know what some of them are. But how do you get the opportunity to work on them? I'll bet they've got designs and code that need reviewing; usability, unit, and automated tests that need writing or reviewing; and bugs that need fixing. I'll bet the developer working on them is out from time to time and needs a backup. But, of course, you're too busy.
- **Customer advocacy and business intelligence** You think you know the customer and business, but you don't. That's the opportunity. The more direct engagement you have with customers (product support, usability, feedback data), and the better you understand the business (VP talks, business plan, business model and metrics), the better you'll know what the killer features are and what the critical features are. But that's someone else's job, right?
- **Critical features** You think these dull features like setup, patching, privacy, compatibility, accessibility, and manageability are for losers. Yeah, they can be when a loser implements them. If you are knowledgeable about the customer and the business, you'll know how to go the extra mile to turn the mundane into the marvelous. But why bother when there are cooler things to implement?
- **Task forces, committees, virtual team projects** You could safely argue that these activities boil up straight from Hades. However, they only arise when there is a problem that someone above your pay grade wants solved. Because everyone hates these dysfunctional efforts, you've got an opportunity to actually lead the effort toward a real solution. Or you could let someone else do it.
- **Process and tool improvements** You probably love these, but no one will listen to your ideas. Stop making improvements your idea. Stop making it about you. Start looking at other people's processes and tools. Start thinking about how you can embrace and extend them. Start thinking about being better together as a team and a company (see "Controlling your boss for fun and profit" in Chapter 8). Or you could just do your own thing.
- **Problems in general** You can hardly make it through five minutes without spotting one problem or another. Every problem is an opportunity. That's not just a cute phrase, it's true. Yeah, you'll never fix them all, but surely there are some worth pursuing. Or you could live with the status quo.

Poor pitiful me

With all these opportunities being there day after day, I'm stunned when engineers complain about the lack of chances to grow and prove their merit. Yes, you are too busy. Yes, you've got enough for your own job. Yes, there are cool ideas to work on that aren't what customers or the business thinks they need. Yes, someone else can run the meeting, write the white paper, or drive the change. Yes, doing your own thing is straightforward because no one else gets in the way. Yes, accepting the status quo avoids hassle. And yes, ignoring opportunities and being mediocre is always the easier option.

"But I don't have time to take advantage of these opportunities," whine the witless. Are you kidding me??!?!? As I described in my column "Time enough" (Chapter 8), you never have enough time to do anything. Each day, each minute, you choose tasks from the unlimited list you have. The key is to prioritize, cutting out the interruptions and time-sucking activities that aren't "must do" in order to focus on the activities that make a difference for our business, our customers, and, in turn, your career.

To cowering clueless who say, "But there's nothing to cut; everything I have is 'must do'!," I say, "So you always accomplish everything you need to do? Really? You must be full of something to say that." Look, if you aren't accomplishing everything, you must be making choices. If you are making choices you simply need to adjust those choices. That's what life is. Successful people make adjustments to focus a portion of their time on new opportunities.

This concept of creating time by choosing your obligations goes by a familiar phrase, "under commit and over deliver." Yet most neophytes over commit trying to impress management and their peers. At the end of the day, these neophytes under deliver, lose out on opportunities, and get ranked below those who meet their obligations and go beyond.

I took the one less traveled by

I'm not saying it's easy. I'm saying it's necessary. No one will serve you success on a platter. Not your parents. Not your boss. Nobody. You've got to decide to take advantage of the myriad of opportunities that come your way every day.

You don't need to tackle all of them, just a few over the course of a year. If you are put on a panel you care about, actively participate. Become one of those people who actually contributes. I don't care if you are busy. Make time. That doesn't mean working longer or harder. It means dropping less important activities that mean more in the moment but less in the long run.

Stop and consider the people you admire. It's not their number of accomplishments; it's the thoughtfulness and impact of their accomplishments. Give consideration to your work choices, be aware of opportunities, create space to take advantage of them, and you can become the person others admire.

Eric Aside As I say above, which some people missed, you only need to pick one to three opportunities to pursue a year. Pick the ones that get your blood flowing. Maybe it's a convoluted process or tool you can improve. Maybe it's a situation that troubled a friend of yours and you think you can fix it. Maybe it's an ignored customer or group issue you want to champion. Go with your passion, but don't go it alone. "Controlling your boss for fun and profit" in Chapter 8 discusses how to suggest a change if there isn't already an effort you can join.

March 1, 2010: "I'm deeply committed"

It's mid-year career discussion time at Microsoft. I could rant about the HR tools we use, but that's like complaining about prostate exams—too inflated a target. Instead, what gushes out at me at this time of year are BOGUS commitments.

You've heard of SMART goals (Specific, Measurable, Achievable, Results-Based, and Time-Specific). BOGUS commitments are Bloated, Outdated, Generic, Unrepresentative, and Self-Centered.

What really kills me are BOGUS commitments posing as SMART ones. They sound specific and measurable. The results seem achievable in the specific time period provided. And yet, these so-called SMART commitments are totally BOGUS. As a manager, BOGUS commitments are particularly gut wrenching to evaluate—"Yes, you are 'On Track' for 11 of your 12 commitments. Unfortunately, your commitments make you look good while your team is failing, 9 of the commitments have changed, and the one commitment that needs improvement is the one that matters the most, but it's lost in the mess. AHHHH!!!!"

How do SMART commitments become BOGUS? Let's break it down.

Eric Aside To its credit, HR has heard all the "feedback" on the performance tools and is actively working on them. Unfortunately, the problems are systemic, so the fix won't be quick. If you're one of the Microsoft employees given a chance to provide constructive feedback on HR prototypes, please do—for the good of us all.

I get bloated with a foamy latte

The first trap for even the SMARTest of commitments is bloat. You've got a whole year's worth of work. The commitments are supposed to be specific and measurable. They are supposed to align to your manager's commitments. Pretty soon, your number of commitments is blowing past 10 and pushing toward 15.

Why is having 10+ commitments a bad thing? Let me count the ways:

1. It takes a long time to write them all. The goal is to get real work done, not to spend weeks writing and editing commitments.
2. It takes a long time for all of them to be reviewed regularly, or discussed at calibration, so they aren't. So they are useless.

> **Eric Aside** Calibration is a process Microsoft uses as part of its differentiated pay system to calibrate expected and exceptional contributions of a group of employees in comparable careers stages and roles. For example, the development managers for Office meet regularly to separate all the Office developers within each career stage into the top 20% of high-potential contributors, the middle 70%, and the remaining 10%. Doing so aligns differentiated pay and our expectations of roles across the company. When used carelessly, calibration can emphasize local versus global optimization, but it's the system we have.

3. They get so specific that the ones for the second half of the year are completely outdated, whereas the ones for the second quarter are only mildly outdated.
4. Since there's no way to distinguish the critical commitments from the "that's part of the job" commitments, you and your manager can't give proper attention to serious problems.
5. Likewise, with countless commitments, it's difficult for you and your manager to highlight your greatest achievements on your review or at calibration meetings.

What's the solution? Have, at most, four commitments (five if you are a manager). Your first three commitments are for your critical project responsibilities this year—the ones your boss will either brag about or have to defend in calibration meetings. Your fourth commitment should focus on personal growth. (Be sure to include mentoring and teamwork skills in this one.) If you are a manager, your fifth commitment is to managing the health and growth of your team. Five commitments. End of story.

> **Eric Aside** What happens if your boss insists that you perfectly align to her 12 commitments? If she is open-minded, you can explain why you'd like to have just 5 commitments, and then align with hers accordingly. If it's her way or else, well there you are. She is the boss. Perhaps you could slip this column under her door.

Living in the past

Plans do change during the year. Your commitments need to reflect those changes or they become outdated. How these changes get reflected in your commitments depends on the type of change.

- A change big enough to alter one or more of your critical project responsibilities demands rewriting your commitments. Obviously, this needs to happen when switching jobs, but it can also happen with a major strategy or organizational shift.
- A strategic change that significantly alters your planned approach to one of your critical project responsibilities requires an adjustment to the execution plan and accountabilities for that commitment. You could leave your commitments alone and just explain the change in the comments, but rethinking your plan and how you'll measure your results is a good idea. Thus, it makes sense to use a commitment change as a forcing function to document and communicate your new approach.
- A tactical change that impacts specific accountabilities for one of your critical project responsibilities could compel you to adjust that accountability in your commitments, but you could also simply comment on it with agreement from your manager. Be sure to get agreement from your manager first.

Eric Aside Revising your commitments with your manager when changes occur, especially big changes like a job shift or significant reorganization, is certainly the right thing to do. It's also seldom done. That's probably because everyone is busily focused on the big changes and not thinking about commitments until the next review period. That's a missed opportunity for setting the right expectations with your boss and the right framing for calibration, but if you make that mistake, at least you're in the majority.

The reason is a little vague

If your commitments and accountabilities could appear just as easily on your peers' reviews as on yours, then they're too generic. Knocking your 10+ commitments down to 4 (or 5 for a manager) might lead you to writing generic commitments. This won't happen if you focus on your three critical project responsibilities, the ones that truly matter this year, the ones you'll never dismiss or hand off to another team.

You can tell if a project responsibility is critical by considering what would happen if you ignored that one responsibility but delivered brilliantly on all the others. Would your manager downplay that one miss, or would that oversight cast a shadow over the rest of your accomplishments?

Each of your three critical project responsibilities has a description, a plan of execution, and specific results that can be measured this year. Write these responsibilities down, and you've got your three primary commitments—the three items you want your manager discussing at calibration.

"What about reviewing other people's code, writing great unit tests, or helping ship our product?," I hear you ask. Those accountabilities are part of being a good engineer. Thus, they form the execution plan and accountabilities for your personal development commitment. It's called the "I'll continue to grow into an impactful and influential engineer for my team and my group who demonstrates the highest standards of quality" commitment. You put all your solid engineer, good teammate, and skill improvement plans for the year into this commitment.

Now when your manager is providing you commitment feedback, she can easily highlight how you are doing in light of your critical responsibilities, whether those responsibilities relate to your three most important projects, your basic behavior as a team member, or your effectiveness as manager of your own team.

I don't believe that's a proper characterization

You want your 4 or 5 commitments to equally and fairly represent your focus for the year at a glance. Having 13 commitments can't help but be unrepresentative.

Why do too many commitments lead to an unbalanced and unfair representation of your annual contributions? Because unequal commitments take up equal space on your review. Yes, that shouldn't matter. You should be able to use bold or highlighters to emphasize the truly important commitments. However, the human mind doesn't care. The text is there, so you read it, and the minor commitments dilute the major ones.

Not all projects are equal. To avoid watering down your accomplishments (or challenges), you need to focus on the critical projects and leave out the minor ones. However, being a good manager and a good team member is just as important as any one project. They each get their own commitment.

As for managers, they should ensure that poor-performing employees don't hide within a herd of commitments. Pull out and bring forward the critical responsibilities that best represent what you expect from your employees for the year.

Eric Aside Many cynical or disillusioned engineers wonder, "What's the point of carefully representing myself in my commitments? My review is determined at calibration meetings, and they happen before my review is written." Most groups work hard to ensure that calibration meetings happen after employees have a chance to comment on their commitments. However, the biggest impact you can have on calibration, beyond your actual performance, is framing the discussion of your performance. Are you working on a critical, difficult, high-risk, impactful, and influential project that aligns with the team's business direction and customer focus? If so, have your commitments say so. That way, your commitments frame the discussion of your work even before you comment on them.

Always thinkin' about yourself

I'm amazed when I read reviews of people who clearly delivered on their commitments, yet their managers thought they were awful. How can that happen? It happens because of poorly chosen, self-centered commitments.

Sometimes the commitments are focused on intermediate personal results rather than final team results. Sometimes the commitments are appropriate for a level below what the team needs from that individual. Sometimes the commitments ignore the importance of collaboration and team dynamics. Regardless, the commitments were written from the perspective of the individual, instead of being written within the context of the team.

If you want to work by yourself, for yourself, then quit and start your own company. If you want to collaborate on projects bigger than yourself for the greater advancement of us all, then write commitments that get you there. Your manager and your mentor can and should guide you through this process. Insist that they do.

Eric Aside Vatsan Madhavan pointed out to me that our HR performance management tool for tracking commitments specifically calls them "individual commitments" instead of something more appropriate, such as "commitments to my team and our customers." Words can be powerful—these words should change.

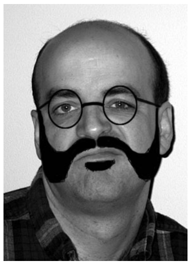
Bogus. Heinous. Most nontriumphant.

No one wants **B**loated, **O**utdated, **G**eneric, **U**nrepresentative, **S**elf-Centered commitments. They don't reflect you, your work, or your importance to your team.

Being SMART is not enough. Don't be BOGUS. Focus your commitments on four or five areas of equal importance. Keep them relevant and oriented toward your impact on team goals. Doing so gives you the best chance to excel at what is truly important to you, to Microsoft, and to our customers.

Eric Aside Whenever I talk to people about commitments, the most common questions I hear are about measuring results and handling stretch goals. I talk about measuring results in "How do you measure yourself?" in Chapter 2. As for stretch goals, remember that missing a firm commitment is underperforming, while making it is achieving; missing a stretch goal is still achieving, while making it is exceeding. Ambiguously mixing stretch goals with firm commitments weakens both—you don't get credit for your stretch goals and you aren't held accountable for your firm commitments. I recommend saving stretch goals for your comments or separating them by using a "meets" section and an "exceeds" section in your accountabilities.

April 1, 2010: "The new guy"



"Hey, you're the new guy!" Marvelous. You've transformed from a useful, relevant, sought-after authority to a roadside attraction. Whoever you were before, whatever value you used to embody, whatever accomplishments you might have achieved now amount to nothing more than marketing hype. Your new coworkers may be outwardly curious and pleasant, but inside they are skeptical and wary.

Look, most people are nice. They want you to succeed. But really, what have you done for them lately? Nothing. After a few weeks, your new-car smell will fade, people will stop treating you like a novelty, and they'll expect results. You've got a lot to prove, yet you know nothing and nobody.

Depending on the job, it will be 6 to 12 months before you're up to speed and can deliver as you have in the past. Meanwhile your productivity drops, your confidence wavers, and your reputation sinks. Marvelous. Yet, you took the new role to enhance your career. What the heck were you thinking? You were thinking that once you got acclimated, life would be better. How can you get acclimated and minimize the collateral damage to your well-being? There are concrete actions you can take.

Eric Aside At Microsoft, we should place more value on what new hires have done prior to their current assignments. By failing to appreciate people's past achievements, we relearn lessons the hard way and discard accumulated talent. Given that the fully burdened cost of experienced engineers is well over $200K a year, that's a pile of money we are composting.

Sure, people can't just rest on their resumes—we need results for today's challenges. However, it wouldn't hurt to learn about people's past experiences and consider their recommendations as a starting point. Everyone could use a head start.

Establishing order

Here's a quick five-step guide to start you on the right path:

1. **Get a grip on yourself.** Understand who you are, what you bring to the team, and your initial and long-term limitations.
2. **Build your support group.** Get to know the key people on the team and value their support.
3. **Extend the honeymoon.** Take care of immediate concerns that will give you quick wins, instant credibility, and breathing room.
4. **Learn the ropes.** Understand the basic workings of the team and get yourself into the flow.

5. **Start your quest.** Pick a relevant and compelling project, and then dive into it, learning all its connections as you go.

We've got our plan—let's put it in action.

Get a grip on yourself

You were hot stuff where you were before. Get over it and get over yourself. A little humility is good for you and will keep you out of trouble. On your new team you have little or no context, so pretending otherwise only makes you look stupid, insecure, or both. You are now resetting into "listening and learning mode." It's good for you.

Accept your initial limitations, and others will accept them as well. While you're at it, reflect on your strengths and limitations in general. You've got a chance at a new start. It's a great opportunity to highlight your advantages and mitigate your weaknesses. ("I can help with that, but please avoid giving me this other thing and having me travel.") When everyone knows what you bring from the beginning, you're far more likely to be set up for success.

"Tell people my weaknesses? Are you insane?!?" No, I'm sane, and you are stupid, insecure, and foolish to think otherwise. Everyone knows you are human—it's not a secret. People's real concern is that you'll screw up the work they give you. They don't know you well enough to account for your limitations. Telling folks about your strengths AND weaknesses up front is reassuring, builds trust, and puts you on the fast track to desirable assignments you can ace.

Eric Aside One combined limitation and strength is your work-life balance. Maintaining this balance limits when and where you work, but it strengthens your ability to reduce stress, provide perspective, and remain a long-term employee. Making your new team aware of your work-life balance boundaries up front is easy and another key to being set up for success.

Build your support group

Getting to know the key people on your new team is critical to your success—relationships matter. Your manager will know who is who on the team, so start there. When you talk one-on-one with these key folks, including your manager, shut your own mouth and listen. (So critical to everything.) Ask questions. Take notes.

Find out about people's roles so you know when to seek their support and on what topics. Understand their goals and problems so you can frame requests around their concerns. Take note of their partnerships inside and outside the team, since these will be the next set of key people to engage. Finally, ascertain what you might be able to do in the near term and long term to help, which will provide the basis for extending your honeymoon and determining your first project.

Eric Aside If you are the manager of a new hire, you can help him or her acclimate by talking about your role, goals, problems, partnerships, and near-term and long-term needs. Then give your new hire a list of key people to meet, and ask him or her to report back with notes. From the notes, you and your new hire can determine what near-term and long-term projects might make the most sense to tackle. You might also suggest one of the key people as a mentor for your new hire.

Extend the honeymoon

Like a new marriage, the first few weeks on a job provide new hires a honeymoon of extra patience, tolerance, and understanding. However, that afterglow fades. Soon folks start expecting you to be the angel they imagined they'd married. Since those are impossible expectations to meet, how can you extend the honeymoon? With quick wins.

By talking to the key people, you know the near-term ways you can help. Many will require little preparation. Taking care of immediate concerns will instantly give you credibility, appreciation, and breathing room.

Quick wins are critical to a strong start on any new team. Utilizing your past skills and knowledge is a great way to build your reputation and create the space you need to fully acclimate. Whether by introducing a process improvement or a nice tool that your old team used, or just being a positive force behind a new initiative or project, you can make a difference quickly. Regardless of what you do, avoid dwelling on your old team—focus on improving your new one.

Eric Aside In my new role, the key people in the group, including my manager, all said, "Bring some stability and a sense of unity to your team." My new team had just completed a round of reorgs. They needed to settle into a rhythm and feel like a team again. This scenario appealed to me when I applied for the role because I have skills and experience in reducing noise and improving teamwork. I defined team aliases and expectations and established regular one-on-ones, team meetings, and morale events. Easy, yet important.

Learn the ropes

Soon after joining a new team, you'll want to subscribe to team aliases; connect to team SharePoint sites, issue tracking, and source control; and get recurring team meetings on your schedule. As you do, take careful note of the team dynamics and basic flow of work. This knowledge will be essential to getting results later, even if those results involve changing team dynamics.

Often teams have a Word document, wiki, or OneNote notebook that details the daily and weekly workings of the team. Reading and updating this information is a great way to learn the ropes of your new role.

Start your quest

After you take care of immediate group concerns, you'll want to initiate a long-term project (a few months in duration). The conversations you've had with key people will provide some suggestions, but your own outsider observations and personal preferences are just as important to consider. To the extent that you have the freedom to choose, select a compelling project that will truly be appreciated by your management and peers, not to mention our customers.

Your first project should be compelling and motivating to you, because it will be tough. You must slog through learning every new tool, system, API, process, dependency, relationship, and spec. The more your assignment intrigues you, the more willingly you'll leap over the hurdles. As you go, you'll learn the connections between people, code, and tools that will make your next project far less daunting.

Asking plenty of questions is a key to success, regardless of how long you've been on a team. However, it's particularly important as you withstand the attack of TLAs (three-letter acronyms), aliases, and team and project nicknames. Be sure to continue documenting the answers you find in your team wiki or OneNote notebook. Remember, everyone knows you are not the all-being—master of time, space, and all human knowledge. Even Doctor Who asks questions.

Eric Aside I've been on a new team for about a year now and followed my own advice. It has worked out quite well. I had two initial projects—improving the deployment process and co-locating the feature teams. I used all the good will I had gained from my quick wins, and the relationships I had developed with key people, to accomplish both projects with great success. It's reassuring when the advice you give others works well for you.

Way to go

Congratulations on your new role. I sincerely wish you the best. Taking on a new challenge is fun and exciting. It's reinvigorating to meet new people, learn new technology, and engage in a new business. To keep that positive energy flowing, you need a plan to make yourself productive as soon as possible.

Reflect on who you are and your strengths and weaknesses. Share that information as you build relationships. Engage the key players within your new team and its partners. Make them happy right away by helping to resolve their immediate concerns. Understand the basic workings of your new team and document them for the next new hire. Choose a relevant and compelling project that will connect you to all the minute details of your new group.

We hire great people, so you must be one of them. A little self-knowledge, a little honesty, a little listening, a little progress, a little learning, and a little perseverance can make for a lot of success.

June 1, 2010: "Level up"

If you're not a Microsoft engineer and you're not interested in finding a new reason to bash Microsoft, save yourself some time and skip this column. If you want to know how to build your skills and systematically grow your career as an engineer at Microsoft, read on.

I've been managing Microsoft engineers for nearly 15 years. I've worked for eight different organizations and more managers than I can recall. While every manager and organization has its own take on promotions, there are some basics everyone follows. Those basics are documented nowhere, until now—right here.

Why am I giving up these "secrets" to getting promoted? Because they aren't secret. My peers all know them, and I've always told my employees this information. After hearing it, my employees all ask, "Why isn't this written down?" I could speculate that HR is too scared to make it simple, that engineers are too precise to generalize, or that Microsoft organizations cling to the notion that they are each special and unique. Whatever, I don't care. I'm just going to tell you.

The basics

Most Microsoft engineers know there are six career stages, each with its own Career Stage Profile (CSP), but they don't know what differentiates one stage from the next.

- **Entry level (e.g., SDE I)** You're straight out of college. Your education hasn't prepared you to work in a professional team environment with money on the line and customers that aren't figments of your professor's imagination. You're clueless.

> **Eric Aside** SDE stands for software development engineer. There are similar stages for the other engineering disciplines.

- **Independence (e.g., SDE II)** You can code up anything your lead asks of you. Period. If you don't know how, you know who to ask, or who to ask who to ask. This is the last time you'll be differentiated by your coding skills.

- **Team leadership (e.g., Senior SDE)** You influence the rest of your team of roughly 3 to 12 engineers. You influence them as their managing lead or as their technical lead. Either way, your impact extends beyond what you can do alone.
- **Group leadership (e.g., Principal SDE)** You influence the entire group of roughly 12 to 80 engineers. You influence them as their discipline manager, group manager, architect, or core technology guru. Regardless, you have become a key person in your organization.
- **Organization leadership (e.g., Partner SDE)** You influence the entire organization of 80 to 500 engineers. You are the discipline director, general manager, partner architect, or core technology worldwide expert. You have become a partner at the company.
- **Industry leadership (e.g., Technical Fellow)** You influence the entire division and with it the entire industry (from 500 to millions of engineers). You are the distinguished engineer, technical fellow, or vice president. Money is probably not a concern of yours anymore unless you have a serious gambling problem.

How do you get promoted from one stage to the next? Let's take that a stage at a time.

Entry level (e.g., SDE I)

To get promoted from entry level to independence, you need to become independent. Duh. It will likely take you two to three years, depending on your past experience. Don't try to save the world. Don't try to impress your vice president. Do your job. Do it well. When something is broken, fix it. Show your manager and your peers that you can handle assignments and get past roadblocks on your own.

Handling assignments on your own doesn't mean doing so in isolation. It means knowing what questions to ask, who can answer them, and how to get the answers. Even the Lone Ranger had Tonto. (I'm dating myself.)

Often, career stages are divided into two levels (in Redmond, SDE I has levels 59 and 60). You'll get promoted to the second level (level 60) after a year or two if you show progress toward independence. You'll get promoted to the next stage (level 61) when you actually are independent, assuming your team hasn't run out of promotion budget.

Eric Aside A common failing at the entry level is complaining instead of solving, aka whining. The world is a messed up place. The workplace is no different. No one else is going to fix your problems. If you've got an issue, then propose a solution to your manager and peers and, ideally, implement it. Otherwise, live with the problem and quit your whining.

Independence (e.g., SDE II)

Now you are independent. You can handle any individual assignment with little direction. You still ask plenty of questions and keep your manager informed, but you are the initiator. No one has to tell you what to do.

Why must you be independent before you get promoted into the independence stage? Microsoft wants you to be successful. If you are promoted and fail at the next stage, Microsoft has lost a valuable employee at the previous stage. Demotion is not an option because it kills morale and motivation. So, **you only get promoted when Microsoft knows you'll be successful at the next stage, which is when you're behaving as if you have already been promoted**.

Microsoft expects every engineer to become independent and a team leader, either organizationally or technically. That means you are expected to make it to the team leadership stage, typically after three to five years of independence. If you don't, you'll likely be managed out of the company. You can't plateau as an independent engineer at Microsoft—you need to influence your peers at least at the team level.

When you have established that you are fully independent, regardless of the assignment, you'll get promoted to the second level within independence. When you are influencing your peers and demonstrating sustained leadership (for 6 to 12 months), you'll likely get promoted to the team leadership stage.

Eric Aside Switching managers and groups often impacts time between promotions because your new manager will not know how long you've been displaying leadership skills. There are a couple of actions you can take to avoid resetting your promotion timetable every time you switch groups or managers:

- ❑ Ask your old manager and new manager to discuss your current leadership skills and how long you've been demonstrating them. This is different from asking for a promotion—you are simply encouraging a smooth personal development transition.
- ❑ If you are going to switch groups and managers, do so within a year or two after being promoted, ideally with the blessing of your current manager. Switching then won't likely impact your next promotion.

Team leadership (Senior)

Sweet! You've made it to team leadership. That doesn't necessarily mean you're a people manager. It means your impact extends beyond yourself and your own work. Your influence guides the work of your teammates (for example, mentoring, setting the example others follow, making broad design decisions that stick, and establishing quality practices that stick).

From here on out, your coding skills no longer differentiate you from your peers. Everyone has great coding skills when they reach the independence stage. Now, your encouragement and influence skills set you apart. Soft skills. There, I've said it. You've got to apply all that written and oral persuasion crap you reluctantly muddled through in high school and college.

Eric Aside Several readers disagreed with my statement that "your coding skills no longer differentiate you from your peers" once you reach the senior career stage. They provided examples of truly remarkable people with extraordinary coding talent that set them apart all the way into the partner career stage. While I don't doubt those examples, I would claim that it was the broad influence of their extraordinary coding that drove their career, not simply that they were prolific. I would also claim that these folks are not the common case.

Microsoft's ranking of the future potential contribution of employees—the top 20% of potential, the middle 70%, and the bottom 10%—changes its character at this stage. That's because team leadership is the first stage where you can plateau, though Microsoft would prefer that you make it to the group leadership stage. The following table shows the differences:

Potential	Entry Level and Independence Stages	Team Leadership Stages and Above
Top 20%	Likely to reach team leadership stage	Promotable depending on progress
Middle 70%	Promotable depending on progress	Promotable, but increasingly difficult to justify because of fewer high-level positions
Bottom 10%	In jeopardy of dismissal unless there is substantial improvement	In jeopardy or reached a plateau—ask your manager to clarify

As for promotions, when you show occasional broad influence within the entire group, you'll likely be ready for advancement to the second level within team leadership. When you regularly and sustainably (for 6 to 12 months) influence the entire group, you'll be ready to reach the group leadership stage. Both promotions can take many years. There is no longer an expected timeline, and no guarantee of a promotion. It now depends on business need and ambition, as well as ability.

Eric Aside What about managing cross-team and cross-division dependency relationships? Isn't that group leadership? No, it's not. Managing relationships across teams and divisions is part of your regular job. Even entry-level engineers need to become good at it. However, when you start influencing strategy and systems of other teams and divisions, you have graduated to group leadership.

Group leadership (Principal)

Wow, you've made it to the group leadership stage as a discipline manager, group manager, architect, or core technology guru. Congratulations, that's a big, difficult step. You got here by thinking and influencing outside the confines of your team.

Influencing a large group necessarily means influencing strategy and systems. I've written many columns on each, in particular "Controlling your boss for fun and profit" (Chapter 8) and "Lead, follow, or get out of the way" (earlier in this chapter). It means thinking big and setting a vision. It means less following, less critiquing, and less complaining. Instead, you step out in front and lead toward your vision. It's scary to stick your neck out, with no one else to cover you. That's what makes you a leader.

You can be promoted to the second level within the group leadership stage simply by establishing yourself as a strong group leader. However, getting to the organization leadership stage is far more difficult. The company doesn't need many organizational leaders. Very few engineers make it past the group leadership stage, so ambition begins to affect your continued advancement. This is true of any sizable company.

There will be peers of yours who have ambition. They'll be willing to live overseas, work on unattractive projects with difficult people, travel frequently and at a moment's notice, be on call at all hours, and make other sacrifices that you may not be willing to make. All other things being equal, an ambitious person will win because of this willingness to do whatever is asked to reach the next stage. If that doesn't sound like you, be prepared to plateau at the group leadership stage. You might make it further, but ambition is a key element.

Organization and industry leadership (Partner or higher)

As an organizational leader—a discipline director, general manager, partner architect, or core technology worldwide expert—you have become a partner at the company. You have special meetings with Steve Ballmer and the executive staff. You get special stock plans and perks because you are responsible for business results as well as work results.

You also are expected to give back to the company. This can mean being moved onto projects that need your expertise, even if you'd prefer to stay where you are physically and organizationally. It can mean being involved in leadership development programs for group leaders, other partners, and new senior hires. It can also mean bumping up against other leaders just as ambitious as you. At this point, you are a grown up. You should be able and willing to handle the politics.

Things only get more competitive to become an industry leader—a distinguished engineer, technical fellow, or vice president. You must be world famous and the voice for a major business that you helped create. Clearly, ambition and business opportunity play key roles in reaching this stage. It's not just about being smart. Everyone who reaches the team leadership stage is smart. You've really got to persist and persevere, doing whatever is necessary, if you plan to be an industry leader.

Set your course

If you haven't realized it by now, career development is a series of choices—choosing to work independently, to influence your peers, to think strategically, and to be willing to make sacrifices. How far do you want to go and what are you willing to do in order to achieve your goals? Those are the key questions you need to answer.

Your career is yours. It belongs to you. Mentors can guide you. Loved ones can support you. But in the end, you own your career. Decide what you want to be. Decide what makes you happy and fulfilled as a whole human being. Accept those choices, set your course, and pursue your dreams. Make it so.

September 1, 2010: "Making the big time"

Review time is almost over. Maybe you got promoted. Maybe your head is filled with thoughts of making it to the big time—calling the shots, getting paid, and having everyone hang on your every word. For entry and independent ICs, that means being a senior or principal engineer (manager or architect). For senior ICs and leads, that means being a principal or partner engineer. For principal and partner ICs and managers, it means being a vice president (VP) or technical fellow (TF).

From the time we are young, western society drowns us with images and icons of successful, powerful, and connected executives controlling their enterprises and living the good life. You grow up wanting that life for yourself—the life you see in movies and media. Your families groom you for that life—telling you to study, sending you to good schools, and encouraging you to be ambitious. Get a grip. Life isn't a movie.

There are great things about being in charge and crappy things about being in charge. Before you go chasing some fantasy, you should know the pitfalls as well as the perks. You should be ready and willing to deal with the crud in order to taste the cream.

It's a question of time

As you increase your scope of influence as an organizational or technical leader, there are many changes. However, there's one dynamic that trumps all the others—the time it takes for your decisions to make an impact grows longer. This is a subtle, yet critical point.

- When you are a college hire, your manager tells you to do something and you do it. The time between articulating what needs to happen and having that happen is usually measured in hours or days.

- When you are a lead telling your reports to complete a project, the time between articulating what needs to happen and having that happen is measured in days or weeks.
- When you are a discipline manager delegating work to leads, the time expected for results increases to weeks or months.
- By the time you are a VP or TF, the time between making any substantial change in direction and seeing the result is six months to three years.

There is nothing you can do about this slowing of sway. Even flattening an organization doesn't help because communication is so imperfect. The more people you influence, the more people who must understand your intent and the longer it takes for everyone to comprehend it and execute it.

The best you can do is state your direction, and then state it again and again—in new ways and through feedback on progress. Remember, most people hold tight to the status quo—you have to continuously nudge folks and sometimes shove and drag them.

The future's so bright

Lengthening the time between decision and effect impacts almost every aspect of leadership. The broader your scope gets, the further out you need to look.

This means that discipline managers and architects need to focus on issues coming next month, not next week. Then they need to direct their teams and remind them several times in advance of the change. Discipline managers who focus on controlling day-to-day issues are called "micromanagers," also known as ineffective, limited, and despised future failures.

For a VP or TF, telling people about what's coming next week is useless. Executives need to be focused on next year at a minimum, while providing feedback and small adjustments to this year.

Blame Canada!

By moving up, you trade fast results for broad impact. You can still get small things done quickly, but you aren't paid to do small things—you are paid for big impact, and that takes time.

This brings us to accountability. If an engineering lead makes a bad decision, the fallout will typically hit within a few weeks. The feedback and correction are quick. If an executive makes a bad decision, it may not have repercussions for years. Those intervening years slow feedback and weaken accountability.

Personally, I like the sweet spot of a discipline manager or architect. Your decisions show results within a few months—fast enough to adjust and learn, yet you still can accomplish great things. In contrast, executives can get away with horrible mistakes for years before being held responsible, assuming they are still in the same division.

Eric Aside Executives who make poor decisions repeatedly are held accountable eventually. Either they fail so spectacularly that it can't go unnoticed, or they leave a pattern of failure in their wake that becomes hard to ignore as they move from role to role.

Of course, you need to find your own sweet spot. If you want instant gratification, you should probably not venture higher than the senior IC or lead role. If you want to help dictate corporate strategy and influence thousands—and you don't mind playing politics and waiting years for results—being a VP or TF might suit you well.

Insects don't have politics

Why do VPs and TFs, as well as GMs and directors, get caught up in politics? Because when decisions are widely separated from results, the right decisions become a matter of opinion. When decisions are a matter of opinion, politics play a significant role. This dynamic is reduced at an engineering company like Microsoft, since many executives are former engineers and insist upon the support of fact, data, and logic. However, the further you get from actual engineering, the more the door opens to personalities, gamesmanship, and alliances.

"But what about history—you know, case studies and best practices?" The applicability of the past to the unique situation in the present is subjective. There are always exceptions and counterexamples. If you are going to be a successful executive, you need to learn to play politics. You must know who trusts whom, who influences whom, who has what agenda, who owes whom favors, who is likely to support or oppose you and why, and what are the hot button issues for the key stakeholders.

It's politics, plain and simple. It appears in every company and government once you get to a certain level of abstraction, where nothing is definitive.

Eric Aside Even worse than politics, delayed accountability can create opportunities for corporate psychopaths—pathological liars with little empathy or conscience. In Chapter 8 of their book *Snakes In Suits*, Drs. Babiak and Hare claim that in working with almost 200 high-potential executives, they found about 3.5% of them fit the profile of a psychopath (around 1 in 30). That's significantly higher than the 1% incidence of psychopathy they found in the general public.

Make it so

Of course, making the big time has plenty of advantages. In addition to the pay, prestige, and perks, you gain enormous influence. Your vision becomes the entire organization's vision—all the more reason to create, articulate, and drive a clear direction. However, even the powerful advantage of influence has its drawbacks.

As a leader, anything you say—no matter how slight—will be taken as direction. Discipline managers soon realize this when they step into their roles. The most minor opinion becomes gospel. It's like a macro version of the Heisenberg Uncertainty Principle—you can't observe a discussion without impacting its dynamic.

Unintended influence becomes a bigger problem as your influence grows. An executive's casual comment often results in person-months of effort. You must be very careful about what you say, when you say it, and to whom you say it. Ideally, you should say the minimum needed in order to guide those around you in the right direction.

Prefacing your words with "This is just my offhand opinion" or "As an ordinary customer, I think..." doesn't help. People still take everything you say seriously. There is no cure—people take things out of context and like to please leaders. All you can do is say what you mean and mean what you say. Otherwise, close your mouth.

Eric Aside If they are supposed to focus on a year out and not talk about much else, how do executives spend their time?

- ❏ The bulk of their time is spent providing feedback, resolving issues, and directing the work in progress associated with reaching their vision. They conceive a vision and then drive it, and drive it, and drive it. Remember, people hold tight to the past.
- ❏ The next large chunk of time is spent building and sustaining relationships with partners, peers, superiors, and the staff. Relationships make everything else possible.
- ❏ Of course, any plan a year out will need to adjust to changing conditions. Replanning is a regular activity, as is preparing for the next big push.
- ❏ Finally, there's the minutia associated with any large role—budgets, resources, busy work, random requests from superiors, events, and personal projects.

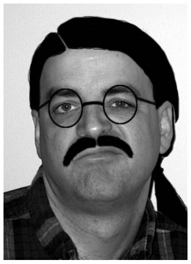
It's alright for you

Now you should have a balanced view of making the big time. Everyone has his or her own limits and views of success. The key is to know yourself and understand what makes you feel happy and fulfilled.

As I talked about in "Level up" (earlier in this chapter), being ambitious means doing whatever it takes to achieve your goals. The higher you want to go, the more of your personal life you will likely sacrifice to get there. It's a choice you should make before the choice is made for you.

I know many executives, and I'm quite thankful they've made the choice to devote themselves to the leadership of our company. I know many senior ICs and leads, who love the jobs they do so well and feel no need to advance further. Now you know both sides. Strive to make the right choice for you.

January 1, 2011: "Individual leadership"



Want to create a noxious gas? Combine ambitious yet clueless engineers, a flat functional organizational structure, and the upcoming midyear career discussions. Soon toxic fumes will emanate from individual contributors (ICs) in response to impotent explanations of upward mobility by overwhelmed managers.

Why the wanton whining from wishful workers? Because the technical leadership path is poorly understood, and the organizational leadership path is narrowing. The flat functional structure of most of Microsoft's divisions today reduces the number of engineering managers and tends to up-level those positions. So now a developer can reach the level of vice president within discipline, but it's tougher for that developer to start out in a lead position.

Functional organizational structures tend to drive most engineers into the IC ranks. The good news is that Microsoft has a robust growth path for IC engineers. At the time of this post, most senior-band and principal-band developers are ICs. It's not until you reach the partner band that most developers are managers. The trouble is most ICs and their managers have trouble describing technical leadership. How do you get started? How do you reach the senior band as an IC? Glad you asked.

There are many ways of going forward

I covered the Microsoft career stages and how to generally move up in my column "Level up." I covered how to get from senior to principal as an IC in my columns "Lead, follow, or get out of the way" and "Get yourself connected." (These three columns appear earlier in this chapter.) The question remains, "How do you first establish your technical leadership and become a senior IC?"

For completeness and clarity, there are three ways to reach the senior band:

1. Become an organizational leader (boss), typically an engineering lead. As I mentioned earlier, these positions are dwindling, especially among the developer ranks.
2. Become a technical leader (guru), typically an expert in some area. This is the focus of the rest of this column.
3. Become a historical resource (sage), typically a longtime team member. This path takes the longest and has the most limited growth potential.

In each case, you are influencing the work of other engineers—as a manager, as a thought leader, or as a resource. Of course, many engineers are combinations of bosses, gurus, and sages. However, being assigned a position as a boss is getting tough, and becoming a sage is a very slow process, so let's focus on becoming a guru.

Ask the expert

Being an effective technical leader means influencing the opinions and actions of other engineers. In science fiction, that influence is accomplished through mind control. In real life, that influence typically starts with expertise.

Basically, you want to become the "go-to" person for some area. The area could be

- A technology, like LINQ.
- A performance area, like power consumption.
- A functional area, like copy/paste.
- A quality area, like accessibility.

The key is that when someone on your team has a problem in that area, he goes to you for a solution.

> **Eric Aside** Most senior and principal engineers have at least two or three areas of expertise but are typically known for only one or two.

I know I've got a bad reputation

Once you've picked an area, you must convince your peers to see you as the area expert. You need to build your reputation around that area—also known as your personal brand. You want to be known as the "LINQ legend" or "power pro."

Your first step is to become well versed in your desired area of expertise. It helps to pick an area you are passionate about and already know fairly well. You can read books and attend talks on the subject, join related distribution lists, seek out other experts around the company, and volunteer to take on significant problems in that area for your group.

Support from your manager can make a big difference. Your manager can help you get the right assignments, approve the right training and conferences, and generally help you to establish yourself. Of course for any of this to happen, you must tell your manager about your interest in becoming an expert in your chosen area.

The next step is being opportunistic. When your area of expertise comes up in conversation (written or verbal), have something intelligent to say that is supported by evidence. "Yeah, Eric Meijer talked about that in his LINQ webcast. The right approach is to..." You can do this in a code review, on your blog, on a distribution list, in a wiki, on Stack Overflow, MSDN, or some other forum—wherever or whenever expertise is needed. The more you contribute in a helpful—not showy—manner, the more people your expertise will influence.

Quality is job one

"But what area should I choose?" Some people naturally migrate to particular areas of interest, but many engineers like technology in general. In addition, often all the "good" areas already have experts on your team. So what's a fledgling expert to do?

Eric Aside If you'd really like to be an expert in an area already taken by another engineer, you can always become the backup for that area. Ask the current expert to be your mentor. Participate in all the design and code reviews for that area. Fix bugs in that area. Cover that area when the current expert is away. Eventually, when the current expert moves on, you'll become the new team expert.

Go for quality! There are plenty of "-ilities" that don't get enough love—maintainability, manageability, accessibility, compatibility, sustainability, availability, interoperability, reliability, traceability, scalability, survivability, testability, and usability, not to mention the big three: world readiness, security, and privacy. Heck, I've probably missed some. Surely, there's an -ility for you.

The great thing about being a guru in a quality area is that the knowledge is transferable to other teams. Many quality areas don't get the attention they deserve, and quality is important to every product. You can continue developing your expertise your entire career, and your personal brand will stick with you wherever you go.

That's a stretch

There are a few pitfalls on the path to expertise. You want your choice of expertise to be your own. Your lead or coworker might suggest a work area to you that appears challenging but really doesn't hold your interest or seems like a death trap. Stay away from that area. Remember, if you are going to be the point person for an area, you'd better like it.

Eric Aside There are some areas, like setup and compatibility, that aren't sexy but are quite complex and particularly valuable. Before you dismiss an area, learn more about it. You may find it's a place you like and can really shine.

Even if you like the area suggested, you need to watch out for stretch assignments. Stretch assignments are terrific—they can really accelerate your career. However, nothing comes for free. If you fail at a stretch assignment, then you fail. You don't get an exceptional review for failing, no matter how much time you put into the assignment. Instead, your reward is extra experience.

Why do stretch assignments? Because the extra experience you receive is extremely valuable. Your career will move forward faster in the long term. In addition, if you are lucky enough to succeed (and there is always an element of luck), then you can get an exceptional review. Regardless, you want to accept a stretch assignment knowing the risks as well as the potential rewards.

What?

Your success in being a respected expert depends greatly upon your ability to effectively communicate your expertise and influence the work of others. If you don't listen well in order to understand people's problems and communicate well in order to describe your solutions, then your vast knowledge is for naught. People might as well consult a brick.

For pointers on listening and providing feedback, try reading "I'm listening" in Chapter 8. For pointers on general communication, try "You talking to me? Basic communication" (also in Chapter 8). The point is smarts will take you only so far. Think of all the successful experts you know—they are all good communicators. Think of all the expert wannabes you know—they all have something in common too.

You can do it

You've got to be a smart, capable engineer to reach the brink of the senior band at Microsoft. The next step as a technical leader is available to everyone. It requires hard work, but it's not especially complicated.

Choose an area you are passionate about. Learn all you can about it. Tell your manager about your interest. Work in that area and on your communication skills. Advise others in that area. Socialize with experts in that area. Become the go-to person for your team.

Soon you'll be spreading your influence across your team and turning heads in the process. It's rewarding, it's fun, and it provides real meaning and value to your work. It's your road to individual leadership—time to buckle up and drive!

Chapter 8

Personal Bug Fixing

In this chapter:

By the time my professional career started, I had already compared academia and the business world, having experienced graduate school and a number of internships. Academia was far more political. In business, raw measures were there to evaluate your effectiveness. I wondered if I was smart and quick enough to compete and succeed in industry.

It took some time for me to realize that once you got past the entry levels, which prune the dim and slow, intelligence and speed don’t make a difference. Your communication skills and how you manage yourself as a human being are what make you effective in the long term. I’ve known smart jerks and quick rogues who initially succeed, only to fall hard when their luck runs out.

In this chapter, I. M. Wright is your personal help desk, providing both the why and how to correct weaknesses that may be preventing you from getting the most from your job and life. The first column reveals the secrets of effective negotiation between people and teams. The second one tackles work-life balance. The third column provides all the tools and tricks you need to efficiently manage your time. The fourth column tells you how to influence those in power. The fifth column breaks down barriers to clear and effective communication. The sixth one exposes the impact of personal and corporate values on our shared success. The seventh tells you why and how to listen. The eighth guides you through the executive review gauntlet. The ninth tells you how to deal effectively with unexpected and *unavoidable interruptions. The tenth consoles you when you've made a bad mistake and makes it better. The last column in this chapter helps you to cope with all the dunderheads around you, including yourself.*

It's easy to blame others for your frustration, poor luck, and the outright unfairness the world throws at you every day. The alternative is to look inside. You can't control what happens to you and the things you care about, but you can always control how you respond. Finding ways to be better every day is the most constructive reply you can give.

—Eric

December 1, 2002: "My way or the highway—Negotiation"

People are paying a lot of lip service these days to cross-group collaboration. Our employee survey results show that almost everyone thinks we should collaborate better, but that almost no one does. Gee, whatever could be the problem?

Could it be that groups have conflicts? Conflicting dates. Conflicting visions. Conflicting features. Conflicting requirements. Conflicting priorities, objectives, customer bases, business models, marketing messages, executive direction, budgetary constraints, resources. Hello?!? For crying out loud, most teams have trouble collaborating within their own group, let alone working with other teams.

Yeah, "But I. M.," you say, "Collaboration is good." Bugs get fixed once—in one place. There's no duplication of effort. The user sees one common way to do things; there's only one user interface to master. Developers need to understand only one API. There are fewer openings for a hacker to attack and fewer holes to patch. Groups can share resources and code. People can focus more on design and less on implementation and test. And, and, and…

So we're back where we started. Collaboration can be both good and intractable, as can working with other people in general. The key is good negotiation skills, something far too few of us know or practice. So what do we do? I've seen two basic methods of collaboration and negotiation used at Microsoft…

An offer you can't refuse

The first way is the "collaborate and like it!" approach, which is by far the most common approach used around here. Someone big, loud, and powerful beats everyone over the head until they work together. (Usually an executive tells the groups to collaborate or else.) When that doesn’t work, the executive slaps the two groups together so there’s no other choice.

This corresponds to the negotiation tactic of yelling louder in an attempt to intimidate people to do things your way. Yuck, how juvenile. No one likes this bullying technique, except occasionally the bully. The worst reorgs are conducted in this way, and they are always nasty. Good people leave the group and sometimes quit the company. Feelings are hurt. Productivity and morale drop like a dot-com stock and take months to recover, if they ever do.

Eric Aside While forced collaboration is still used, it no longer dominates at Microsoft. These days, teams have agreements with each other as I describe next, or they simply share source code. It’s far better, but people often don’t understand the reasoning behind the agreement and skip important aspects. This predictably leads to problems, which is why knowing how to negotiate is so fundamental.

Grow up

The second method is a more mature approach to collaboration. It is also a bit subtler. You form a kind of agreement with your partner group. (Yes, I know there are obsessive PMs out there who mess this up by writing detailed and demeaning contracts that demoralize and disengage their dependencies, but you needn’t overdo it like that.) Up front you simply agree to what your group and your partner group will do, what your group and your partner group need in return, and what your group will do if things don’t work out and likewise for your partner.

This method is analogous to the effective negotiation strategy of discovering and disabling threats while fulfilling needs, thus establishing trust between parties and creating a basis for compromise and collaboration. That was a big sentence with lots of big ideas, so let’s break it down.

A shadow and a threat have been growing in my mind

When two groups are okay with the idea of mutually benefiting from working together, the big problem becomes removing barriers, not the collaboration process itself. This philosophy isn’t true if the groups want to defeat one another, like we might feel toward our competitors.

However, for groups within Microsoft or when working with our partners, the real challenge to collaboration is getting all of the obstacles out of the way.

Don't shoot the messenger

Barriers to good collaboration always come in the form of threats, needs, and trust. Remove the threats, fulfill the needs, and you establish trust. Everything is downhill from there.

- **Threats** So, say you would like to use Windows Messenger in your application, but you are afraid that the Messenger team's schedule will not match yours and that they will make last-minute changes that cause your program to break or your project to slip. This is a valid threat.
- **Needs** You need the Messenger team to grant you a stable code branch around your ship dates, and you need them to verify their builds around your usage of their APIs.

  On the flip side, the Messenger team has needs to be met—they don't want you to cause them grief—no huge feature changes or requests, no additional localization costs. And they want you to use their setup module so that you don't break the other applications that use Messenger. They need you to agree up front to use the component as is, to cover any additional loc costs, and to incorporate their setup module in your application.

  Eric Aside Truly understanding and appreciating your partner's needs and concerns are critical to successful negotiation. Make sure you are in sync by directly acknowledging each other's situation and requirements. No compromises yet, just acknowledgment will go a long way to building trust.

- **Trust** The contract you make, which can be as informal as an e-mail, documents that your team agrees to use Messenger's setup and cover the cost of any additional loc; their team agrees to give you a frozen code branch around your ship dates (easy with Source Depot) and to use your BVT to verify their drops. You also specify what happens if either of you becomes dissatisfied with the relationship. Then product unit managers (PUMs) from both teams agree to the terms, and you are set to work together as partners.

  Eric Aside A Build Verification Test (BVT) checks if a software build satisfies a set of requirements. Using a BVT to check software drops from other teams before accepting them is a wonderful practice. Even better is giving your BVT to the other team so that they can check themselves. That way, they know when they have met your requirements and you never have to deal with an unacceptable build.

After that, you can work out drop locations and schedules and solicit help implementing the APIs. This becomes easy when you both trust that your concerns will be met.

So happy together

The key to maintaining this level of trust is to keep the communication lines wide open. Ensure that PMs from both sides are talking to each other about schedule or feature changes, problems, and surprises. It's not hard, but if either team forgets, it can spell doom for the project.

Eric Aside A mnemonic the Harvard Negotiation Project recommends is ACBD, Always Consult Before Deciding; that is, don't make any impactful decisions without first talking to your partners.

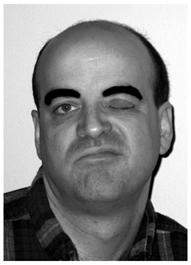
When you negotiate this way, by removing threats, fulfilling needs, and gaining trust, you become incredibly effective in all facets of life. A mistake that people often make when negotiating is to jump in with solutions before listening to the concerns of others. If you prematurely present a solution, no one will accept it; they are afraid to get hurt. If you listen well, ask questions, and resolve the issues first, people will be surprisingly open to your ideas.

Eric Aside Be sure everyone feels like a winner after the negotiation is complete and beyond. Losing is a universal threat. One of the easiest ways to ensure everyone wins is by including everyone on the winning team—for example, "Here's OUR great design."

This method of collaboration is not magic and it's not just for working across groups. Good negotiation skills can help you at home, around the neighborhood, within your own team, and with our partners. So get out there—collaborate and like it!

Eric Aside "You can depend on me" in Chapter 2 talks more about managing dependencies.

February 1, 2005: "Better learn life balance"



Warning: Mushy, reflective column follows. Proceed at your own risk.

I didn't want to work at Microsoft. I was happy with my previous job, and they treated me well. Working at Microsoft would mean giving up my life and my family, working outrageous hours, and putting the company before all other things. I had many friends who worked at Microsoft. They often asked me to join them, and I always said "no."

Then I decided I needed a change and entertained an offer from one of Craig Mundie's old groups. I wasn't a young, campus hire. I had a wife, a home, a two-year-old son, and another boy on the way. I had no intention of giving that up. I told my prospective boss that I was a family man. I would only accept a position in which I saw my kids off to school every morning and ate dinner with them every night. To my surprise, he agreed heartily. More importantly, he kept his word.

Eric Aside Craig Mundie is now Chief Research and Strategy Officer for Microsoft.

Balance is key

The subject of work-life balance comes up all the time in discussions that I have with devs around the company. Heck, it comes up with everyone I meet. Many employees I've spoken to genuinely feel that they must choose between Microsoft and their personal lives. The only disparity is the degree of commitment demanded.

That is tragic. Not just on a philosophical level, but often on a personal level. I've seen people get divorced, lose custody of their children, become ill or depressed, and let go of friendships and all semblance of a personal life. I've seen it happen to friends and coworkers from the time I joined the company nearly 10 years ago until within the past year.

Eric Aside The real tragedy, as I describe next, is that all this grief is unnecessary and not what Microsoft or any decent company wants for its employees.

We are whole human beings. Denying work-life balance denies our selves. There are three ways we commonly cause ourselves grief:

- **We let work trump everything else.** As I just mentioned, the consequences are clear and devastating.
- **We have different values at work.** Our values define who we are, and trying to be two different people is wrenching.
- **We try to keep work and home separate.** Living a double life is stressful and impossible to manage.

Words without action

Our leaders have said that balance is important. Part of Bill Gates' message about changing the world together was "for everyone at Microsoft to develop a challenging career with opportunities for growth, competitive rewards, and a balance between work and home life." Steve Ballmer has talked about the importance of his own home life, especially as his family has grown. Yet this message apparently hasn't become a reality for many employees.

Eric Aside As I mentioned in the first chapter, Steve Ballmer, our beloved CEO, practices work-life balance himself. I've met him several times while he was cheering on his son at a basketball game or going out to a movie with his wife.

It is easy to put much of the blame for our lack of work-life balance on management. Although almost anyone would say balance is desirable, it is sometimes hard for managers to juggle balance with other business priorities. Even managers who support work-life balance can send contrary messages unintentionally.

For example, during a crunch time, a manager may start ordering dinners for those who "choose" to stay late. So devs who arrive at 10:00 A.M. will stay until 8:00 P.M., working a 10-hour day. But their teammates with kids arrive at 9:00 A.M. and only stay until 6:00 P.M. (nine hours). Then from home, they log on at 9:00 P.M. and work until midnight, working a total of 12 hours. The devs who "left early" feel guilty about abandoning the team, don't get a free dinner, and actually work longer.

This contrived example isn't meant to knock anyone; it's meant to demonstrate a reasonable case where hard-working people can be penalized unintentionally. If managers wish to promote a fair and balanced work environment, they must apply a fair and balanced reward system.

But the responsibility for work-life balance doesn't end with management. Bill's next sentence asserts, "In a fast-paced, competitive environment, this is a shared responsibility between Microsoft and its employees." Given management's focus on results, is taking personal responsibility for work-life balance reasonable or career suicide? Within a certain framework, I have found balancing work and home to be both achievable and respected. If you think my claim is fanciful, wait until you hear about the framework.

I can't even balance my checkbook

Achieving balance isn't easy for your checkbook, let alone your life. Here's my five-step program for an advantageous balance between work and home:

1. **Understand and accept your lifestyle choices.** The first step is to know yourself. What are your priorities? Does career come before home? What are your limits? Would you give up a parent-teacher conference, but not the school play, to advance your career? You must understand and accept these choices, even if they seldom arise. This will prepare you to speak with your manager from a position of strength and conviction, one he or she will respect and uphold.

   Eric Aside My experience has been that this step is the hardest. Most people have never had to confront their life choices. Being honest with yourself and truly deciding where you draw the line between work and home can be challenging, but it's extraordinarily important and valuable.

2. **Set ground rules with your manager.** Every time I begin reporting to a new manager (a common occurrence at Microsoft), I discuss my work expectations: "I see my kids off to school every morning and eat dinner with them every night. If that is unacceptable, I will respectfully seek a different position." We always agree that there will be occasional exceptions, but the ground rules are clear and established. No manager has turned me down, and sticking to my ground rules has not impacted my advancement. However, jobs that require lots of travel are not for me. Many of my managers have told me that they consider my strong convictions and clear values a strength.
3. **Do not compromise quietly.** Occasional breaks in the usual routine at work are expected, but a two-week trip to Japan would be a big deal for me. When this type of request comes up, I take the opportunity to reassert my constraints. Often an alternative is available; sometimes I just need to go. Either way, my manager is reminded of my priorities and his or her commitment to honoring them. If you compromise too easily, it tells your manager that you don't really care that much. Your manager will likely continue to ask for more and more of you until you finally do care, establishing a new, less desirable limit.
4. **Use RAS and Remote Desktop or OWA as needed.** Of course, there are crunch times several weeks every year. Before terminal server (Remote Desktop), crunch times meant that I returned to work after putting my kids to bed. These days, I frequently log on from home after they go to sleep—not because I'm always in crunch mode, but because I love my job and like the work. But I spend just as many nights watching TV or movies with my wife. I choose my activities based on creating the balance my family and I need.

Eric Aside Remote Access Service (RAS) is a means to tunnel into the Microsoft intranet from home. Outlook Web Access (OWA) is an AJAX application that permits you to access your corporate e-mail, calendar, and contacts from any Internet connection.

5. **Drop the schizophrenic pretense of separation.** For years I had a work life and a home life. That's how I was always told it should be, regardless of how awkward or uncomfortable it got. Now I just have a life. Period. A family health crisis seven years ago forced me to drop the absurd pretense that I could separate work and home. No one can effectively separate their lives in two, unless they have a serious personality disorder. So, live the same at home and at work, within certain limits of decency and responsible behavior.

Balance good...everything good

Being in touch with what we need to live a full life and then living by those standards are wonderful gifts that we can give ourselves. Part of this is graying the line between work and home so that we aren't constantly context switching our values and our souls. This doesn't mean that we spend all our time at work chatting with friends and relatives any more than it means we spend all our time at home working online. It also means that we must honor the privacy and needs of our coworkers. Remember, the Microsoft value is open and respectful, not just open.

But if you do integrate work and home, a beautiful thing occurs. Lessons that you learn at home help you at work. Insights you gain at work improve your life at home. Balance yields tremendous personal growth as well as personal well-being. You may never become a megalomaniacal industry magnate, but the great riches you gain will not be so easily lost.

June 1, 2005: "Time enough"

"Work smarter, not harder." Wouldn't you like to shove cliché-spouting, nonsensical know-it-alls into a meat grinder to give them a more representative look? Especially when the message comes from a middle manager who has assigned 18 "high priority" projects to your team and then joins the rioting mob of inane, inconsiderate ignoramuses who constantly interrupt you to ensure that you can't possibly think.

There is never enough time to keep up with even a fraction of the work that is typically on your team's to-do list. Even if you had a reasonably scoped set of work, the relentless interruptions and meetings would still make completing any real work laughable.

Yet, this is the job of a typical dev lead or manager. Good managers get through each day successfully restraining themselves from strangling half the people in their hallway. Great managers learn to manage their time.

Give it to me straight

There are countless books and organizers for time management. Most seem to me to be embarrassingly superficial, wildly impractical, or written for aliens from a parallel universe where anal retentiveness is an indicator of high social status.

For me, time management comes in three basic varieties:

- Cutting down interruptions and context switches.
- Delegating your work to other team members.
- Careful selection of work items based on priority and leverage.

Pardon the interruption

To be effective, you need to concentrate. Recent studies have shown two interesting findings:

1. Heavy context switching causes a higher drop in IQ than using marijuana.
2. Employees are most productive when they are working on precisely two projects.

The first finding shouldn't be surprising. You can't think if you can't focus due to constant interruptions and context switching. The second finding is more subtle. It says that you need to concentrate on one project at a time, but need a different project to switch to when you get stuck or need a break on the first. The second project is treated as filler—that is, useful but not essential.

Some interruptions are easy to control. I turn off all forms of e-mail notification and I set the ring volume on my phone to its lowest possible level. (The phone just quietly clicks.) As a result, I no longer get interrupted by e-mail or the phone. I reply when I'm ready to take a break, not when someone else happens to click the Send button or dial my number. This does not make me less responsive than I was before; it just puts me in control.

When I do find a good stopping point and scan my e-mail (usually every 10–40 minutes), I try to completely dispense with each message as I read it. Each message represents a context switch. You reduce context switches by reading each message once. (The Lean folks call this "single-piece flow.") More than 95% of my e-mail can be deleted, cataloged into a folder, transferred to someone else, or responded to immediately upon receipt.

Eric Aside For more detail about my single-piece flow solution to e-mail, read "Your World. Easier" in Chapter 3.

Some might claim that this process puts undue priority on minor items. However, you must read a message to determine if it's minor. After you've read the message, it usually takes more time to context switch back and re-read it later than it would to simply dispense with it now. As a bonus, when your inbox is almost empty, it takes far less time to find specific messages and see what's left to do.

As for Instant Messenger, I don't use it at all. I believe IM was sent to us by the devil to enslave our teenagers and ruin our lives. Of course, that's only my opinion.

Eric Aside My older son is a teenager now. I stand by I. M. Wright. To my friends on the Messenger team, "No offense."

Find your happy place

Another way to reduce interruptions is to get lost. Go somewhere that no one can find you, and finish some work. Armed with remote desktop and a laptop or kiosk, you can work almost anywhere: meeting rooms, building lobbies, cafeterias, you name it. As far as anyone can tell, you're in a meeting. You can't do this all the time without harming your team, but it's great when you need to catch up.

You can also work when others aren't around. Most developers show up around 10:00 A.M. and work until 7:00 P.M. If you show up around 8:00 A.M., you have two uninterrupted hours. Leave at 5:00 P.M., and if necessary you can log on from home after 8:00 P.M. when things are quiet. I wait until my kids are asleep if I've got extra work to finish.

Finally, you can institute focused time during the week when people are expected to not interrupt you or your team unless absolutely necessary. For this policy to be successful, you must choose a predictable time when interruptions are less likely anyway. One day or an afternoon and evening during the second half of the week is often a good choice because most meetings and fire drills happen early in the week.

None of us is as dumb as all of us

An especially evil form of interruption is the meeting. A meeting forces you to stop productive work and throw yourself into a frustrating, time-consuming, black hole of wasted life from which you can never recover. However, there are several actions you can take to minimize the life-draining effect of meetings:

- **Stop going.** There are only a few meetings you must attend: your one-on-ones, your project status meetings, and your staff meetings. Almost all other meetings are discretionary. If a meeting feels optional, try skipping it. If nothing bad happens, don't go again.
- **Make someone else attend.** Try delegating the meeting to someone else. (More on this in the following section.)
- **Run effective meetings.** As for meetings that are left, make them as effective as possible. Read my column "The day we met" (which appears in Chapter 3) for tips on running tight meetings.
- **Put all your meetings back to back.** I know this sounds strange, but the idea is to reduce context switches. First, try to schedule your project and staff meetings early in the week, and then schedule all your one-on-one meetings around them. Sure, the early part of your week will be hellish, but the middle and end of your week will have uninterrupted blocks of time.

A burden we must share

Although cutting down on interruptions is great, the most effective way to get your work done is to give it to someone else. There's a reason why the lives of leads and managers are so busy—you've got a whole team of people to look after in addition to your project work. The balancing factor is delegation. Managers can leverage their teams to lighten their loads.

Eric Aside Architects can use architecture teams this way too, even though the members of the team don't necessarily report to the architect.

There's a tendency for new leads and managers to avoid burdening their teams. They'd rather take on the stress and load than seem weak or lazy. That's foolish, cowardly, and selfish. The worst thing you can do to your team is stress yourself out. When you are stressed out, you become short with people, you don't take time to think and listen, you make bad decisions. Soon your team follows your lead, becoming stressed and dysfunctional.

You must be the rock for your team. You must keep your head when others lose control. You can only do this when you work within your limits. Hand over all the assignments you can to your team. Remember, they only wish to support and please you.

Why does your team want to support and please you? Duh, you write their reviews and initiate their promotions. They need challenging assignments that expand their scope to get that 4.0 and advance their careers. They need the kind of assignments that you're asked to do. Why hold your people back? Why hoard all the tough, critical tasks? Delegate your work to your team, and give your people what they want and need to advance.

Eric Aside The 4.0 refers to a high rating in the old Microsoft rating system, which ranged from 2.5 to 4.5 (the higher the rating, the better the rewards). While a 3.0 was acceptable, most people pursued and received a 3.5 or higher.

Tell me what I must do

When you delegate your work, it's important to do it right. The trick to delegation is to delegate ownership, not tasks.

The difference is subtle, so I'll give you an example. Say your project depends on Media Player and you're scheduled to meet with the Media Player team. You don't feel like going, so instead you plan to send Anil, a dev on your team who wants to become a lead.

If you just delegate the meeting to Anil, you know what will happen. No matter how well you prepare him, Anil will be asked to answer questions and make commitments he isn't equipped to handle. He'll meet with you later to recap the meeting. You'll ask all kinds of questions that he can't answer, and you'll both be left with the feeling that you should have attended the meeting yourself. The result is frustrating for you and feels like failure to Anil.

Now, let's say that instead of delegating only the meeting to Anil, you delegate the whole relationship with Media Player. You bring Anil into your office and say,

> *Anil, I want you to OWN our relationship with Media Player. You ensure that they have our technical requirements and that we can commit to theirs. You own working with their team, understanding their APIs, and designing and coding our end of the interactions. How does that sound? Oh, and by the way, the first meeting is coming soon.*

Anil will love it. He'll go to the meeting empowered to answer questions and make commitments. He'll feel a sense of authority along with responsibility. Anil is set up for success. Meanwhile, not only did you get out of that meeting, you got out of every future Media Player meeting. What a difference.

He's just a kid

If Anil is a junior person and delegating ownership is risky, there are several ways you can mitigate the risk:

- Accompany him to the first few meetings, but keep your mouth shut; defer to Anil as much as possible, then debrief him later.
- Assign Anil a mentor who can play a similar role to the one I just mentioned.
- Ask a friend on the Media Player team to look out for Anil and let you know if there are any problems.
- Ask to be on the e-mail threads sent between groups, and have Anil provide regular, detailed status reports.
- Use some combination of these methods.

Regardless of how you proceed, Anil owns the relationship with Media Player and has the opportunity to show his stuff while leaving you more time to focus in other areas.

You deserve a break

The ownership approach to delegation works well for almost every type of assignment. Any time you have a task to hand out, even if it's only to cover for you while you're at the dentist, stop and think about the larger context for the task and assign ownership of the whole scope.

By the way, if you're having trouble getting started with delegation, take a two-week vacation. You'll be forced to delegate all your work—and when you get back, you can let folks keep it. Plus, you get a two-week vacation as part of the package. Sweet!

Everything's in order here

The last major variety of time management is how you select which work items to pursue and in what order. Start by listing all your current tasks and decide which ones to keep.

The work items should fall into a few categories (listed in priority order):

- Tasks that require your personal attention as part of your job (taking care of your team, reviews, one-on-ones, staff meetings, employee issues).
- Tasks that are critical to your personal development goals (training, key assignments, executive or customer engagements).
- Tasks that enable you to stay engaged with your team members and manage them effectively (morale events, project and team meetings, design and code reviews, debugging, triage, and hand-picked project work).
- Tasks you happen to particularly enjoy.

All items in the first category must be on your list. Often there are tasks that overlap between categories; these are high-leverage items and should definitely be on your list. Everything else can potentially be dumped or delegated.

After you've pared down your list of work, you need to order it. Consider the priorities of the categories I mentioned, how leveraged the task is, and the urgency. In the end, you should have only two or three major projects and a few minor items.

Remember to focus on only one major project at any given moment and give it your full attention. Switch to other projects when you get stuck or need a break.

Keeping it real

One of the most highly leveraged work items you can have is direct project work, like owning, designing, and coding a feature. Project work gives you insight into employee issues (build problems, team personalities, cross-group interactions), assists in your personal development by keeping you sharp and current, and engages you with your team and larger organization in a direct and integrated fashion. You'll almost certainly enjoy it, assuming you pick the right task.

Some people debate whether or not a dev lead or manager should still code. To me, direct project work is so beneficial to yourself and the team that you can't afford not to do it. The trick is picking the right task. If you have three or fewer reports, you should have enough time to devote to almost any project assignment. However, with four or more reports, people and project issues will arise frequently enough that your availability will be unpredictable.

Having an unpredictable schedule isn't a problem early in a project. However, as you get closer to the end, your peers across disciplines will demand delivery commitments that you cannot keep with certainty. You'll end up handing unfinished work to your team members, who are already under pressure to complete their current assignments. Plus, you won't have time to transition your work sufficiently, which makes the transfer even more painful. Now your peers hate you, your team hates you, and you hate yourself because you had to give up and give in.

All this changes if you choose project work that isn't critical path. In particular, with more than three reports you should select work that

- Doesn't need to ship.
- Can be easily disabled.
- Is risky, fun, cool, and an unexpected delight to customers.
- Ideally, is well integrated into the product.

When you select an assignment with these attributes and your peers across disciplines demand delivery commitments, you can say, "Honestly, I think I can finish it in the time frame you need, but I can't make promises. However, if I don't get it done in time, I can easily disable it and it won't ship." Now, your peers are comfortable, your team is unaffected, and you are stress free and feeling fulfilled. Oh yeah, it's the only way to go.

By the way, it's not hard to find this kind of project work. Usually risky, fun, and cool features that are well integrated into the product are too scary to make critical path. So you can assign them to yourself and enjoy keeping it real. Management does have its benefits.

Large and in charge

It's easy to let a lead or manager job take control of your life. The interruptions, overwhelming commitments, and disengagement from "real work" can make even the most capable people dream of when they first started working and everything seemed so simple.

However, there is a variety of ways to take back control of your work load, reduce interruptions, create opportunities for your staff that lighten your load, and trim your assignments to only those that provide the most benefit to you and your team. Applying these techniques puts you back in charge. That's exactly the place an effective and engaged leader needs to be.

August 1, 2005: "Controlling your boss for fun and profit"

There's a great gesture you can do to show just how little you care about someone who is wallowing in self-pity. You lightly rub the tips of your thumb and forefinger together saying, "This is the world's smallest violin playing, 'My Heart Cries for You.'"

That's how I feel when people complain about their helplessness in the face of the seemingly invincible power of their manager. "Oh, management will never give us the time to improve our build or change our practices." "I wish our manager took this training. That's the only way we'd ever take up inspections." (See "Review this—Inspections" in Chapter 5.) "I'm really uncomfortable with our product's current direction and group's organization, but there's nothing I can do."

Grow up, you weenies. Based on your pathetic excuses for inaction, nothing would ever get done. Don't you think your bosses say the same thing regarding their bosses? If you don't make desired change happen, it doesn't happen. Period. The difference between you and someone powerful is not your level of control; it's your willingness to act.

I have no hand

"Yeah, but my boss won't listen to me," is the common retort. "She has all these reasons why we can't change." Well, good. At least you've made the transition from being pathetic to being ignorant. You're trying to do something about your situation; you're just inept. Relax, most people are.

Unfortunately, influencing without authority rarely comes naturally; it is an acquired skill. When trying to enact a solution, most people jump right in. They go straight to their bosses with their idea, only to get shot down. Sometimes people even do a tremendous amount of preparation, writing a long white paper or presentation, only to be summarily rejected.

What you may not understand is how to prepare appropriately, how to present your idea effectively, and how to get your idea enacted. Let's break it down.

Eric Aside I gave an internal talk on this topic. Registration filled in minutes. The moral: many people want to know more about influence without authority.

Know the enemy and know yourself

Start with appropriate preparation. I'm going to list a bunch of steps, but they can all be done in less than a day (minutes for small issues).

First you need to scout the landscape. Only fools walk into a minefield without a map. Most people know how they want things to be, but that's an endpoint, not directions to get there.

- **Understand your proposal.** What is risky about your idea, and how do you mitigate the risks? What can change about your idea, and what are the core principles you can't compromise? Be realistic and honest with yourself.
- **Understand your history.** What were the reasons for your current processes and organization? Could you regress by changing them?
- **Understand your enemies.** Who prefers the current state and why? Are any of them strong enough or passionate enough to make change difficult? How would you placate them or even draw them to your side?
- **Understand your friends.** Who is unhappy with the current state and why? Do they like your idea? How strong, considerable, and passionate is your body of support?
- **Understand your management.** How is your management being measured or judged? What benefits would be worth the risk from your manager's point of view? Can you increase the benefit to management or reduce the risk?

It sounds complicated, but if you have been paying attention to how your coworkers interact, you and a friend or two can scout the landscape in a candid short discussion. Talking to a few people is often necessary to unearth the history or understand the issues. Regardless, scouting is essential to success.

They succeed in adapting themselves

Now that you know what you're up against, you need to adapt and refine your original idea accordingly:

- **Choose who to please, who to placate, and who to ignore.** Sure, you want to please everyone, but sometimes it's better to just make sure nobody gets upset and focus on the few you really need to make happy—like your boss. Some folks will be fine if you just keep them from harm. Others can be ignored if they follow whatever your boss decides or simply don't care.
- **To please people, focus on the benefits in their terms and negate the risks.** Use your scouting information to frame the benefits in ways that impress the key players. If your manager cares about efficiency, talk about productivity gains; customer satisfaction, talk about quality and connection; on-time commitments, talk about predictability and transparency. To negate risks, talk about backup plans, go/no-go decision points, redundancy, and/or clear prioritization.

Eric Aside This is the negotiation step of removing threats and fulfilling needs I talked about in "My way or the highway—Negotiation," which appeared earlier in the chapter.

- **To placate people, neutralize whatever they find threatening.** Use your scouting to uncover what's scary. If it's risks, negate them. If they have a competing solution, embed it and give them ample credit and the limelight ("It's our idea"). If they have extra requirements, satisfy them either immediately or in the "next version."

In the end, you'll have a plan with a bunch of backers, no one will be apprehensive enough to fight it, and you'll be aligned with what the key decision makers care about. Now you're ready to present it.

Selling water to fish

Selling water to fish really isn't that hard, assuming their credit is good. You simply need to show them what life is like without water. The same thing goes when you are driving for change. You need to frame the problem before you talk about your solution.

The focus is on the key players, usually management. In the same way you use your scouting information to frame benefits in key player terms, you frame the problems in terms that speak to key player concerns. This should be the first slide after the title in a very short deck.

A few important notes here:

- **Anything about you or your ambitions will poison the proposal.** The proposal needs to be about the key players, the team, and the customers, not about you and your desires for fame, glory, or a high rating.
- **To target your presentation on the key players, you must slip inside their skin.** Use their terms, talk about benefits to them, and address their concerns. While the solution may have been spawned by what you care about, the solution will belong to the team, not you. It will be owned by the key players, not you. You must leave your feelings out of it.

  Eric Aside Generally speaking, slipping inside the skin of key players and leaving your own feelings behind is the hardest step. It's also incredibly important to success.

- **If you talk about the solution first, no one will care.** If you skip over the problem, there is no impetus to change. If you talk about the solution and then the problem, people will fixate on the problem and forget your solution. Always start with the problem and then move on to talk about the solution.
- **You must keep your presentation short—very short.** Change drives discussion and debate. The debate will expand to fill any allotted time. If you can't get through your ideas in two or three slides, you'll lose all sway over the argument.

Eyes on the prize

Your second and possibly third slides speak to your vision of the future state. The vision should be clear, concise, and practical. The goals should be clearly stated in terms of benefit to the key players. Simply put, you cannot reach a destination if you don't know where you're going.

You'll likely have many more slides with all sorts of data and detail about your vision and proposal. You might even have a 30-page white paper. That material is important to document the basis for the change, but relegate supporting material to appendix slides and resource links. You must be crisp to be successful; everything else is there for reference only.

Your last slide addresses how you get from the current state to the future state, or how to reach your destination. There are only two sections to this slide:

- **The issues section**, which addresses how risks and concerns are mitigated. It's where to put your backup plans, go/no-go decision points, redundancy, and/or prioritization.
- **The next steps section**, which addresses who does what and when. Too often people focus on what needs to be done and not enough on who will do it and when it will happen. Without specific people assigned and specific target dates, the change flounders.

That's it: a title slide, problem statement, future state, and transition slide. Now you are prepared to bring your idea forward. For smaller ideas, you can do all this in an e-mail message, but the preparation is the same.

Eric Aside A number of people doubted someone could put all this information on three slides and asked me for an example. My best example was a Microsoft confidential proposal that put all the information on one slide. It had vertical and horizontal lines dividing the slide into four quadrants: problem (four bullets in the upper-left quadrant); solution (three bullets in the lower-left); issues (six bullets in the upper-right); and next steps (four bullets in the lower-right).

Engage

You are now ready to engage with the key players. There are dozens of ways to do this, and none is dramatically better than the others. Sometimes the landscape may suggest one approach over another. In general, there are three basic types of approach:

- **Talk to key players one at a time.** This works well when the key players don't get along or each has different key issues. It's more time-consuming, but it's usually a safe and effective approach.

- **Meet with the key players together.** This works well to build consensus and bring out hidden issues. It's also a bit faster, but it works best if some consensus already exists about the problem and the issues. If necessary you can start with the first approach to gain that initial consensus, and then seal the deal with a meeting.
- **Target only the top player.** This works well when the top player is particularly strong or the organization is particularly compliant. Use this approach when no one really matters but the top player.

Go through your deck and be prepared to own the process you've created. Remember that it's not about you—it's about the team, the product, and the customer. Let people discuss and debate as much as they want, as long as they stay focused on the problem you identified. If new issues or risks arise, be sure to note them and devise mitigations.

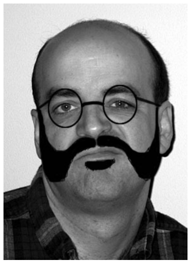
Eric Aside "The VP-geebees" later in this chapter goes through the particularly tricky realm of presenting your proposal to executives.

Dare to dream

This whole process may seem like a great deal of trouble, especially when there's no guarantee your solution will survive, let alone thrive. You may feel that it's not worth it; that the status quo is acceptable or at least tolerable. Maybe that's true, or maybe you're spineless.

Just don't start complaining about how powerless you are or how management won't listen to you and doesn't care. If the current state is acceptable, then accept it and move on. If it isn't, then do something about it. Regardless of what happens, you will drive awareness of the problem and likely cause change. In addition, you'll gain leadership experience and perhaps even gain leadership responsibility. In the end, you will become more powerful by matching your willingness to act with the courage to focus on your team instead of yourself.

April 1, 2006: "You talking to me? Basic communication"



I read a lot of e-mail. I go to a lot of meetings. I read a lot of code and specs. I go to a lot of reviews. I read a lot of white papers. I go to a lot of presentations. Aside from getting the life sucked out of me, I've realized something: most communication is a terrible, tragic waste of time.

That's surprising because the principal difference between junior and senior engineers is their impact and influence. Because everyone is smart around here, the primary driver of impact and influence is strong communication skills. You'd think people would get it right. Of course, I think that about most things.

I've railed against long meetings (in "The day we met," which appears in Chapter 3), poor specs (in "Late specs: Fact of life or genetic defect?," also in Chapter 3), poor spec reviews (in "Review this—Inspections," in Chapter 5), and unfocused presentations (in "Controlling your boss for fun and profit," earlier in this chapter). Yet, communication is incredibly important for collaboration, growth, and teamwork. Surely between the days of ancient Egypt and now, people have learned how to effectively converse. But where is the evidence? Our e-mail and meetings are bloated and pointless, our code and specs are indecipherable and inadequate, and our white papers and presentations are self-serving and self-indulgent.

Eric Aside Naturally, I'm overstating the problem again; there are many examples of great communication both inside and outside of Microsoft. However, if raising the alarm means more concise e-mail, clearer specs and code, more engaging presentations and papers, and shorter meetings, I'm all for it.

Why? What's so hard? Where is the source of the problem that takes what precious time we have and fritters it away so carelessly? After many years of contemplation, I believe at last I have found the answer: people aren't thinking enough about me.

Think about me

Yes, people aren't thinking enough about me, the receiver of their mammoth, malformed musings. What is everyone told the first time they write a paper or prepare a presentation? Consider your audience. It's elementary, yet we still have severe signs of self-serving spew. That's because while people do think about me, their audience, they don't think about me enough.

Traditionally, considering your audience has meant understanding who they are and what they know so you can target your communication appropriately. However, apparently that's vague and insufficient. Here's a concise, complete list of what you should consider about me:

- What do you want from me, specifically?
- When do you want it from me? Do I have a prayer of meeting that?
- Why should I care? Will I pay attention long enough to even listen?

Eric Aside For this entire column on communication, I'm using "me" to stand for your audience, however large or small it might be.

Tell me what you want

Good communication starts with knowing what you want from me—not what you want in general—I couldn't care less. What do you want from me?

Does that sound callous? Heck no, it's as open, respectful, and honest as you are likely to get. We all have hopes, dreams, and ambitions. It would be fun to discuss yours and mine over beers sometime, but right now I'm at work and I'm busy, so try to stick to the point.

Do you just want to keep me informed? Save it. If your information doesn't have a purpose, then I don't need to hear it. You say it's important that I know? Why? What actions do you want me to take later based on the information? How should I use it?

Seriously, what do you want from me? If you don't know, then I certainly don't. And if neither of us know, then we're just wasting each other's time.

Put what you want up front, in the first few lines, the first few slides. Be clear, not bashful. Use bold and my name to highlight what concerns me. I really want to care, but I can't until I know what to care about.

You want it when?

Do not bother asking me for something I can't provide. It's insulting to me and useless to you. I charge big money for miracles, so seek them elsewhere. Don't only think through what you want, think through when you want it and what impact that time frame has.

Sometimes your time frame is long. In that case, don't ask too soon because you'll only have to ask again later. For e-mail, I can simply delete pointless requests; but premature meetings, specs, code, and presentations can really cost me time, not to mention all your wasted effort.

Sometimes your time frame is short. Don't pretend it isn't. I don't mind people asking me for favors, I just mind them taking me for granted. Know when the ask is big and appreciate it when people come through.

Sometimes your time frame is a mystery. If you don't say when you want it right up front, you might as well have not asked in the first place.

Got a short little span of attention

Now that you know what you want from me and when, how about we stick to it? Let's face it. People's attention spans are short and I'm no exception. You need to grab my attention and keep it till I both understand what you want from me and I care enough to actually help you.

Here are techniques for grabbing and keeping my attention and for making me care:

- **Be concise.** Provide essential context, but get to the point. Don't let my mind wander. If it's e-mail, put everything in the first three lines (the ones that appear in auto-preview). If it's a meeting, make it short with a small agenda. If it's code, keep the function on one screen. If it's a presentation, use one slide per concept and only provide details I care about, not what you care about.

  Eric Aside An example of "put everything in the first three lines" of an e-mail: "Mr. Galt, though you make good points, your last few e-mails have been overly long. Please shorten your communication to a paragraph or less by next week."

- **Be focused.** Stick to the point. (Note the "die" in digress.) Stay on the agenda. Invite only those you need to a meeting, and include only those you need on an e-mail thread. That means don't "Reply all" without editing down the list. Keep function code, white papers, specs, and presentations coherent. If you are taking me down a path or building up to a climax, it had better be worth it. Just know what you want from me and be singularly focused on it.
- **Be simple.** If I have to squint or read it twice and your name doesn't rhyme with "fates" or "calmer," then your cause is lost. Three bullets per slide, five pages per paper, one idea per function, one decision per e-mail, and one feature per spec. Use pictures and stories; pretend I'm five years old. If you can't explain it to a five-year-old, then it's doomed to failure and isn't worth my time.
- **Be organized.** E-mail should read like a news article, with the overview and ask up front, followed by detail as needed. Longer documents should tell a story or follow some other familiar pattern. (Steal from good examples.) Meetings should have an agenda. Presentations should tell me what you're going to tell me (outline), tell me (tight slides), then tell me what you told me (summary). Dull? Only if you make it that way. Add zingers and funny pictures, and be passionate and playful, but stick to the structure. There's a reason why we have these story patterns—they work.
- **Be respectful.** Don't ask obvious questions that anyone with a web browser could answer. Anticipate objections and questions, and respond to them before they are raised. Don't pretend you know something that you don't, especially laws and patents. If you don't know, say, "I don't know." Choose your words carefully when communicating directly to customers, competitors, executives, and people who may be feeling a bit emotional. When presenting, don't go over your allotted time and do leave time for questions. Don't read your slides, and don't tell someone they asked a good question—I can read for myself, and my question was just as good.

- **Be smooth.** Use proper grammar and spelling. Have someone review your e-mail, particularly if it's sensitive or if you are, let's say, emotional. Use clear variable names and common terminology. Practice your speeches—it's a physical activity. Take steps to relax before you present. There's no presentation hall without a bathroom—use it for quiet time and mother nature, then focus on what you want from me and make me believe. At the end of your speech, conclude clearly with, "Thank you, any questions?" and accept the applause you richly deserve.
- **Serve me.** Communication isn't about you. You already know what you know. Communication is about me, your audience. Tune your message to my concerns and the type of information I care about. If I ask for data, give me the facts and spare the stories about your grandma. If I have a "bad feeling" about your idea, then skip the data, show me a demo, and have your grandma reassure me. If my boss won't like it or it doesn't fit our process, then convince my boss or change my process and stop wasting my time. Remember, different people are convinced in different ways and care about different things. For more pointers, read my column, "Controlling your boss for fun and profit," which appears earlier in this chapter.

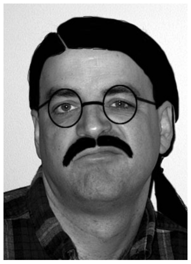
Are we done?

Communication is a big subject, and I'm covering a lot of ground here. Take your time, practice your communication skills, and get good at them. Microsoft is full of smart people like you. To truly differentiate yourself, learn to communicate effectively.

Thank you, any questions?

Eric Aside And thanks to Jim Blinn, for his advice and closing line from "Things I Hope Not to See or Hear at SIGGRAPH." (You can find an excerpt at *www.siggraph.org/s98/cfp/speakers/blinn.html*.)

March 1, 2007: "More than open and honest"



Our corporate values are slightly off. I can't disagree with them. It's hard to argue against integrity, honesty, passion, openness, respect, challenge, self-improvement, and accountability. Those are all good things, to be sure.

But are these corporate values the right ones for Microsoft? Say our contact with a key dependency, we'll call him "Deadbeat," tells me his team dropped a critical feature a month ago because their priorities didn't match ours. Is it okay because they are being "honest"? Do I feel consoled because their triage was "open"?

No, Deadbeat is dead meat as far as I'm concerned. I don't care if his team is truthful about breaking its promises. I'm not impressed that their catastrophic decisions are made in public. Our team is now knee deep in sewage with no nose clip and little time to recover.

That's no excuse

People use Microsoft values to excuse themselves from guilt. Deadbeat says, "I know we agreed to ship that feature, and I respect your passion about it. It was a challenging decision, but we're accountable to our dates and had to cut somewhere. I'm just being honest with you. Yeah, we should have told you earlier, but the triage meeting was open. I guess communication is an improvement area for us."

Now, don't you feel better? They were just exemplifying Microsoft values. Whoop-de-do. We are still in deep trouble with no advance notice. What went wrong?

You could say it was a lack of accountability, but they were accountable to a date and a set of priorities. Those other accountabilities just trumped ours.

The problem is more subtle and has a serious impact on how we develop software, collaborate across teams, and run our business. It starts with our first value, "Integrity and honesty."

Eric Aside You can find more effective ways of managing dependencies in "You can depend on me," which appears in Chapter 2.

I'll be honest with you

People think integrity and honesty are synonyms, but they're not. Honesty means you don't lie. Integrity means your beliefs, words, and actions match.

Eric Aside I later found a marvelous quote comparing honesty and integrity. "Honesty is…conforming our words to reality. Integrity is conforming reality to our words." —Stephen R. Covey

You can be honest with no integrity ("Yes, I'm the one that back-stabbed you."). You can be dishonest with strong integrity ("I'm okay with postponing the bug, I understand."). Personally, I respect honesty, but I value integrity. You can believe someone who's honest, but you can count on someone with integrity.

Deadbeat and his team were honest but lacked integrity. They agreed to ship the feature, but didn't follow through. When they needed to cut our feature, Deadbeat didn't contact us. He didn't offer us alternatives, redress, or an apology. He didn't take his word seriously.

It’s not easy

Honesty is easier than integrity. No one can blame you for being honest. Integrity requires courage and conviction. You must risk upsetting others, including your management and your peers, in order to stand by your beliefs. It’s far easier to compromise your standards, assuming you have any. That’s why integrity is so valuable.

It’s not easy, but when you demonstrate your personal integrity in difficult situations you gain people’s admiration. They may not agree with you, but they will know you are a person of character and substance; a person not easily bought or manipulated; a person to be respected.

Honesty is not a strong enough value upon which to hang our partnerships and our business. We must have integrity. Compromise on prices, plans, and particulars, not on principles. When you give your word, keep it. When you can’t keep your word, apologize, then make it right.

They seem to have an open door policy

But integrity is not enough to make cross-team collaboration and partnerships work. You need transparency. Transparency doesn’t even appear on the company values list.

People think open and transparent are synonyms, but they’re not. Open means you’re public, you have no secrets. Transparent means you share the who, how, what, when, and why of your decisions and actions.

You can be open without being transparent (“I don’t know why we made that decision, but the meeting was open.”). You can be closed yet be transparent (“The negotiations were behind closed doors, but this is our consensus decision and here’s why we made it and who was involved.”). Personally, I value transparency far more than openness. You have access to someone who’s open, but you can rely on someone who’s transparent.

Deadbeat and his team were open but not transparent. They didn’t share how or why they cut the feature we needed. In fact, they didn’t share the decision at all till it came up a month later. Had we known when the decision was made, we could have at least argued our case and then adjusted our plans.

Eric Aside Naturally, consulting all your partners before deciding, or even informing your partners promptly about decisions, is easier said than done. You must have a list of partners at the ready, and a process for notifying them that doesn’t get bogged down or bottlenecked. I recommend the SCR process described in “Late specs: Fact of life or genetic defect?” (which you can find in Chapter 3).

No place to hide

Openness is easier than transparency. It's safer to conduct yourself in public, where social norms protect you. Transparency exposes your weaknesses. You don't get to pick and choose what you will share. You must own up to your true situation and risk disapproval. It's far easier to be open and hide behind, "All you had to do was ask." When you are transparent, everyone always knows the score. That's why transparency is so valuable.

It's not easy, but when you are transparent even in crisis you gain people's trust. They may not be happy with you now, but they will never doubt you. You become known, understood, and reliable.

Openness guarantees communication can happen, but not that it will. Transparency ensures our teams and our partners have the information they need to deliver on commitments. Make your schedule and status public. Share your bug queries, acceptance criteria, and build results with your partners. Celebrate promotions and successes, and talk candidly about failures and necessary improvements. Give those who work with you reason to believe in you.

Eric Aside We've had an interesting internal debate recently around making people's career stage transparent in the e-mail address book. On the positive side, promotions, role models, and who you are calibrated against become public. On the negative side, a class system is exposed, entry-level opinions may not be respected, and senior people may get differential treatment. It's a tough call, but groups are tending toward transparency, which at least exposes the class system so that impropriety can be handled appropriately.

Not what I had in mind

You're probably saying, "Of course, integrity and transparency are important, but so are honesty and openness. Aren't you making a big fuss over minor word differences?" Don't be so sure. Remember, I said the problem is subtle and has a serious impact on how we develop software, collaborate across teams, and run our business.

When people think honesty and openness form the bar, bad things can happen. While I've never regretted upholding my integrity or being transparent, honesty and openness can cause trouble even with the best of intentions.

Honesty can be cruel and heartless, but more importantly it can be deceptive and dysfunctional. That's because people often don't know the real truth, just what they currently believe. They can speak with sincere conviction yet be completely misguided. This leads to misunderstandings, wasted effort, and often animosity.

Openness has the same kind of issue. People don't do everything in public for a reason. How you behave depends on who is there to see. I behave differently in private with my family or my team than I do in public meetings or talks. It's not that I have something to hide; I don't. It's that my family and my team have context that the general public doesn't. That's why negotiation must be done in private in order to be effective. It's intimacy and confidentiality that breed candor and flexibility.

Getting it right

My plea for integrity and transparency doesn't mean I'm against honesty and openness. These are all exemplary values for us to follow. In particular, openness also means being receptive to the thoughts and ideas of others. The world would benefit greatly from an increase in that kind of openness.

But being honest and open falls short of what we need to be successful and can sometimes lead to unintended trouble. We need the integrity to mean what we say, say what we mean, and honor our words and beliefs with our actions. We need transparency in our decisions, our policies, and our projects to allow us to operate and collaborate knowing where we stand.

These may be small requests, but they have huge impact. There are people at Microsoft who feel threatened by transparency or lack the courage to uphold their integrity. They should look deep inside and change either who they are or where they work. For Microsoft to continue as a great company, our values need to mean more than words on a page.

March 1, 2009: "I'm listening"

It's midyear career discussion time at Microsoft. Perhaps you just finished, but more than likely you're still trying to squeeze yours in. How'd it go? How will it go? For you? For your manager? Well, that depends.

It depends a bit on your prior performance and your manager's prior performance. It depends a bit on the feedback itself and how that feedback is given. It depends a bit on how your parents raised you and the comfort of your chair. But the biggest influence on the lasting impact of your midyear career discussion is the way you and your manager respond to feedback.

Let me put this delicately to you. You have no frigging idea how to give and take feedback. Seriously—not one frigging idea. Think I'm wrong? You are only proving me right. If you actually knew how to give and take feedback your response would be a sincere and polite "Thank you."

Thanks for the advice

In fact, there are only two valid responses to feedback: "Thank you" and "Go on."

The "Thank you" is simple and self-explanatory. Too bad most people don't use it. Most people defend themselves, explain their behavior and results, and describe how they are already taking the right steps.

Please, slowly and carefully shut your mouth, empty your mind, and listen. Perhaps you can even take notes. Then, when the generous soul is finished, say "Thank you."

You don't say it to be polite. You say "Thank you" because you mean it. Your relationships, your life, and our products and services would reek far beyond their current stench if people were not kind enough to provide an outside perspective and help us improve. Thank goodness they are willing to do it. To ensure they continue it's essential to sincerely appreciate it.

Eric Aside What do you do if the feedback is self-serving or perhaps political and unconstructive? You say "Thank you" and "Go on." You say "Thank you" because the person is letting you know how they feel about and approach the situation—in poker parlance, they are showing you their hand. You say "Go on" because maybe the person is willing to give away even more of their strategy and inside information about those you might be up against.

Tell me more about me

In addition to "Thank you," a valid response to feedback is "Go on." As in:

- "Could you talk more about that?"
- "I don't quite understand—could you describe that further?"
- "Thank you, that's helpful, what else can I do differently?"

Anything that encourages clear and continued feedback is appropriate.

Back off, man. I'm a scientist

What's inappropriate is anything that questions or cuts off feedback. This includes:

- "I'm working on it." So what? You're doing the wrong thing, you haven't made much progress, or you are actually improving. Regardless, the feedback is valid and your comment is irrelevant and self-serving.
- "I was trying to…" So that makes it better? Never confuse reasons with excuses. If you can get better, you should get better. No excuses.

- "I disagree." So this is news? You're getting feedback. It's opinion. The fact that you like your current approach is not a revelation. When the feedback seems wrong, you've either missed something or left the wrong impression—that's precious information.

Keep in mind that you don't have to follow whatever advice you get. All you are obligated to do is listen, consider the advice carefully, and thank the person for helping you.

> **Eric Aside** A particularly important time to keep your mouth closed, take notes, and simply say "Thank you" is during an executive review. You can learn more about executive reviews later in this chapter ("The VP-geebees").

Now it's my turn!

Now that you know how to take feedback, it's time to learn how to ask for it and provide it. When asking for feedback and when providing it, there are three basic questions:

- What is good [about what I'm doing or the work I've done]?
- What could be better?
- Any further comments?

You can structure feedback more, but the simplest, complete ask are those three questions. And it is those three questions that you want to answer when you provide feedback.

> **Eric Aside** Feedback is best provided immediately before or after behavior. The ideal is to provide positive feedback directly after desired conduct and corrective feedback just before it's needed. In other words, apply feedback at precisely the moment it is most constructive.
>
> For example, a guy on your team sends a great e-mail but forgets to copy a stakeholder. You reply to him right away, "Great mail—concise and insightful." Later, just before he's supposed to send the next update you write, "Remember to copy all stakeholders." The reminder is more useful at that time.

We have come full circle

When providing your feedback, start with what's good, talk about improvements, add on your other comments, remind about improvements, and then reiterate what's good. That order is important.

- You start with what you like. It sets up the conversation on an upbeat note and prevents the impression that all is lost. If you start with what's wrong, your listener may never hear what's right.

- Next, you talk about ways to improve. Ideally, your listener should focus on just one change. One change is all that most people can handle at a time. Pick the most impactful improvement and emphasize it.
- Of course, you'll have plenty of other thoughts that aren't as important. Feel free to mention those in the context of "a few other comments."
- Then come back to your main message—the one or perhaps two improvements that would make the most difference.
- Finish with what is going well. It's important to end on a positive note.

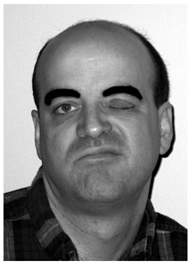
Eric Aside Remember to always focus on the behavior or outcome, not on the person. People can't change who they are, but they can improve their actions and results.

We don't have much time

That's it. Being concise is important. If you want your feedback to matter it should be clear, consumable, considerate, concise, and centered on the receiver. Your feedback isn't about you and your glorious knowledge; it is about helping the recipient.

If you are on the receiving end of the feedback, you should be just as concise. Feedback is precious, whether it's from a customer, a peer, or your manager. Don't get in the way. Encourage it. Savor it. Appreciate it. Thank you.

July 1, 2009: "The VP-geebees"



It's the end of the fiscal year. Most engineers associate this time with performance review season, but for principal-level engineers and higher it's also executive review season. Time to waste weeks of your life writing slides for executive presentations that will be rewritten five times before they are never presented.

Executive reviews aren't a waste of time—occasionally you need an experienced, authoritative voice to blow apart your assumptions and refocus your efforts on desired business results. Preparing isn't a waste of time—being forced to explain yourself to others always helps your thinking, and I've got no desire to look like an idiot in front of the person who approves my compensation. The real waste of time is focusing on getting the right slides for every season and situation instead of getting the right strategy.

Smart, high-level people simply don't know how to cope with executive reviews. They think it's a time to show off instead of a time to listen. They respond inappropriately to executive criticism of their badly presented, unsuitable slides. I've done it too—it's a trap set by our superiors filling out the poor templates dictated by their superiors. It's the misinformed leading the uninformed. Well, now it's time to break that cycle, avoid the pitfalls, and focus on what matters—valuable feedback on your clear and concise plans.

Wisdom to know the difference

Why do so many otherwise intelligent people bungle executive reviews? I believe there are two reasons—exuberance and confusion.

- **Exuberance over the importance of the moment** It's so important that we must cover every detail rather than focus on what's important. Twisted huh? Who defines what's important? The executive. What's important to the executive? Ask. Press for clarity. Don't accept second-hand smoke from an assistant. It's not good for your health.
- **Confusion between executive reviews and presentations** Both use slide decks. Both involve presenting. The difference is that in presentations the presenter is in control. In executive reviews, executives and their posses are in control. Failure to recognize this difference leads to failure in your review.

The secret of my success

How do you handle executive reviews successfully, obtaining all their potential benefits? Here's a three-step guide:

1. Learn what is important to your executive.
2. Present that information in three slides or less.
3. Respond to questions with insight and to feedback with thanks.

That's it. Let's break it down.

A riddle, wrapped in a mystery, inside an enigma

You first must learn what's important to your executive, from both an informational and a philosophical perspective.

From an informational perspective, what does your executive want to know about your project? Alignment with other projects? Financial contribution? Market share impact? Value proposition? Competitive response? You want to learn how your plans are being evaluated.

From a philosophical perspective, what principles are most important to your executive? Transparency? Alignment? Loyalty? Integrity? Self-confidence? You want to learn how you are being evaluated.

How do you learn your executive's informational and philosophical perspectives? Ask peers who've already been through a review with your executive. Ask your boss and your skip-level boss. You can even get thirty minutes on your executive's calendar. Be sure not to believe any one individual, but see patterns in feedback from multiple sources.

Eric Aside It's also good to know the executive's mood in advance. While we might hope that mood wouldn't influence executive decisions, executives are human beings, and adjusting for their mood can have an impact.

Easy as 1, 2, 3

Now you are ready to create the slide deck. You need only three slides—the current situation, the desired situation, and the tactics to get from the first to the second. That's all you should have. Remember, you are not in control of the review—the executive is. All you can hope to do is frame the discussion. All the other slides can either be cut or left as appendix slides for reference.

Eric Aside For deep technical reviews, you might be tempted to have more than three slides—resist that temptation and relegate all the deep technical information to the appendix slides. You'll likely cover a number of those appendix slides, but the executive drives the discussion not you. You just need to be ready to address the executive's questions and issues.

The current situation slide may be a problem statement, a current scorecard, or a recap of progress to date. The proper context and information depend on what's important to your executive, which you learned earlier.

The desired situation slide may be a solution, a target scorecard, or a going-forward strategy. It should align with your first slide and resolve the issues it raised.

The tactics slide may be a timeline, bullet list, or table of next steps, typically also indicating risks and mitigations with associated asks. Careful what you say on this slide. It tends to convert directly to commitments.

Eric Aside The tactics slide seems like it covers a lot of information. Of course, all three slides do. You need to understand how your executive likes information presented.

- ❑ Does she like eye charts with all the information on the slide?
- ❑ Does she want summary slides and expects you to have all the details in your head?
- ❑ Does she prefer detail slides in the appendix that she can reference?

I read the instructions

Many executive reviews require you to follow a template with predetermined slides for you to fill-out. Having a template is terrific—it adds consistency and clearly sets expectations. Unfortunately, most templates are hideous with four or five times the number of slides needed. The templates are either ancient or produced by assistants.

How do you handle a fifteen-slide template? Pick out the three crucial slides—the ones that describe the current situation, the desired situation, and the tactics to get from the first to the second. Get those three slides right—then ensure the other slides align consistently with your crucial slides. This keeps you on message no matter where the executive takes the conversation. If possible, skip past the other slides and stick to the crucial three.

The purpose of the review is to get valuable feedback on your clear and concise plans. By aligning your slides and staying focused, you can frame the discussion and get the input you need.

Eric Aside A handy tip is to send out a preread three days before the review. Write a one-page document that addresses the executive's key questions or provides context that attendees should know before the meeting.

The preread helps the conversation stay focused. It also gets the executive in the right frame of mind and provides the opportunity to send clarifying questions to you in advance so you can better target your presentation.

Oh behave!

The last step in my three-step guide to executive reviews is about how to behave. First and foremost, do not be intimidated or enchanted by the executive. Executives used to have your job and they miss it (just ask if you don't believe me). They still use the bathroom. They still embarrass their kids. Do yourself a favor and get over yourself. Act like you've done this before.

The executive will typically take two actions during your review—ask questions and make comments. Most of the time you want to take notes and keep your mouth firmly shut. Use duct tape if you must. Abraham Lincoln said, "It is better to remain silent and be thought a fool than to open one's mouth and remove all doubt."

When should you say something? When you have something insightful and relevant to say. "I agree" doesn't cut it. "We're doing that" doesn't cut it. "When we fixed that issue our support calls dropped 67%" might be worth mentioning if it's relevant. When you open your mouth, be sure you are adding value, staying respectful, and being concise. "Yes," "No," and "I don't know" are often sufficient. The rest of the time you should listen.

Eric Aside If you need help being succinct, tell a friend at the review to give you a signal when you need to stop talking—perhaps a flashing neon sign.

As I mentioned in "I'm listening" (earlier in this chapter), the single proper response to all feedback is "Thank you." In the case of executive feedback, be sure to write the comments down and consider each point. You don't have to address everything the executive mentions, but you do need to consider it.

Remember, executive reviews aren't the time to show off. They are the time to receive valuable feedback on your plans. The time to show off is when you deliver on those plans.

How'd I do?

When the review is finally over, you'll likely feel awful. The executive asked many tough questions and made a bunch of pointed comments. Was it a disaster? How can you tell?

Luckily, it's easy to tell how the review went. The key is the level of detail the executive discussed, not in the number of positive or negative comments. If the executive's questions and comments were all general and high level, then the review did not go well. The executive was questioning your basic strategy and assumptions. If the questions and comments were in the details, then the review went quite well. The executive agreed with your strategy and approach and was giving you feedback on small pieces.

Either way, executive reviews aren't about giving you an ego boost. They are about getting valuable feedback on your plans. Learn the information and principles the executive cares about, present your plans as simply as possible within that context, and be a professional during the review, and you'll get all the feedback you need to succeed.

Eric Aside Are executive reviews really necessary to operate a business? At times they seem to act as a crutch for a lack of organization and aligned planning. When an executive staff's plans are aligned and the organization is built to match the plan, then all the executive should need to run the business are regular staff meetings and coordination meetings with executive peers. It also helps if the executive is accessible to everyone in the organization through blogs, one-on-ones, events, and casual encounters. A number of Microsoft organizations are now run this way.

December 1, 2009: "Don't panic"

In my last column, "Spontaneous combustion of rancid management" (Chapter 9), I talked about how managers should restrain themselves from randomizing their employees. But what if you are on the receiving end? As an employee, how do you best respond to a random request, or requests of any kind, that aren't directly related to your current task?

Sure, we'd all love to be the guy or gal in charge and not have to answer to anyone else. That's how many startups get started—people want to be their own boss. As if being the CEO helps you avoid random requests from customers, clients, and creditors. Look, you can't escape it. Even Steve Ballmer's day is dominated by crud to corral, catalog, and contain.

What kills me is that everyone seems to believe their own lives are busy but that everyone else is idle and waiting to serve them. That's stupid. We're all busy—really, really busy. At times insanely busy and stressed out. So when that next request comes and you have no time, but it's the boss or the customer or the partner who can't be ignored and you have to deal with whatever it is—even though your brain is vibrating, your inbox is full, and your stomach is aching—what do you do? First of all, don't panic.

You should say yes

You respond to the request by saying, "Yes, I'd be happy to help." Counterintuitive, isn't it? After all, you can't possibly help. You don't have the time, nor does your team or anyone you know. The key is learning how to say yes the right way. (Later, I'll talk about the few well-defined situations when you should say no.)

Why say yes to almost any request regardless of your situation? Because saying no doesn't fix the situation. It just gets worse. Ignoring the request is no better than refusing it. The most effective response is yes, framed in a few different ways.

- "Yes, tell me more about your problem." Most requests are unclear. Maybe they are nothing at all. Maybe they weren't meant for you. Almost certainly, the people on the other end haven't thought it through. Providing an initial "Yes, tell me more" only commits you to learning more. After the request is properly defined you've got an easy exit as needed ("Oh, you meant that?"). In the meantime, you've made the request go away for a while—sometimes forever.

Eric Aside Why don't you ask why first? You are asking why, but in a different way. Why might cause a defensive response, whereas "Yes, tell me more about your problem" is less threatening and more likely to illicit the information you need.

- “Yes, let me point you to a person or website that can help.” Many requests go to the wrong people or can be served by an online resource. Passing those requests on appropriately fulfills the need. Be short and clear to avoid answering again. Don’t worry about how busy the next person is—they can say yes just like you.
- “Yes, when do you need that done?” Clarifying when a request needs to be filled transforms it into a prioritization exercise. Given enough time, you and your team can do anything. Given high enough priority, you can do anything right away. It’s all about priorities and timing. Let’s drill down on that.

Eric Aside Why does the situation get worse if you just say no? Because the request lingers. It becomes more urgent. By the time it pops up again, it’s even more difficult to manage. Meanwhile, you look unresponsive and uncooperative to your manager and customers. You’re much better off handling it right away.

All other priorities are rescinded

You ask when the request needs to be completed and naturally the answer is—”Right away!” Now what? It depends on who’s asking. If it’s not the people who control your schedule, like your management chain, you say, “Great, let me pass this request by my manager to ensure it gets done.” Sometimes even threatening to run a request by a manager is enough to make the request go away—if not, it’s time to talk to the boss or project manager.

At this point, your management is making the request or you are escalating a request to management. Either way, the conversation is the same. You present your prioritized work backlog up to the cut line for the current milestone or iteration (a whiteboard will serve). You ask where the new request fits into the prioritized list. There are only three possibilities:

- The new request fits below the cut line. You reply to the request, indicating it will have to wait till at least the next planning period. If that’s a problem, refer the requester to your manager. This keeps you and your team on track and focused on your top priorities.
- The new request fits above the cut line. That means the lowest priority items move below the cut line, which you point out to management at the time of the decision. You communicate the change to everyone impacted and let the requester know when to expect action.
- The new request fits above the cut line, but your manager wants nothing else cut (sound familiar?). You negotiate the scope of your existing work items. Perhaps some can be simplified or reduced. Perhaps other people can take items off your list or help you handle the new request. Don’t leave the conversation without a mutual agreement and understanding of the changes, and then immediately communicate those changes to the appropriate people in writing (that makes it stick).

Of course, you may actually want to perform the new request. So long as it doesn't impact your commitments, you can go ahead and do it. However, don't kid yourself. If it will take you more than a couple of hours, you owe it to yourself to formally adjust your schedule. No one will talk glowingly about all the work you did while you failed to do your job.

Eric Aside Note the importance of passing a prioritized work backlog up to the cut line for the current milestone or sprint. If you are using a method like Scrum, this is trivial. Even decent waterfall project management methods use prioritized task lists. Regardless, make sure you've got one. It's a great bonus if your manager is already familiar with it.

Trust but verify

Now you are extremely busy and have a new task. Perhaps you negotiated with your manager for help from other people. Unfortunately, those people are clueless. They're not going to do it right, which is to say they're not going to do it the way you would. What do you do?

You delegate ownership of significant portions of the request—not tiny disconnected work items, but ownership of significant portions. You say, "You own this piece. Here are the requirements. Here are the boundaries. Here are the people involved. Here are the results I expect and when I expect them. Any questions?"

The bottom line is that people will not do things the way you would, so let go. Give them ownership. Let them discover the right way for them. It will be better than your way for them. That's how you delegate. You trust people. Seriously.

Naturally, you want to regularly check people's progress and results. Trust but verify. That's the key to delegation and management in general. Trust your people, verify their work.

Know when to say when

There are times when you should say no or when you can ignore a request, but there are only a few well-defined situations.

- If the request was sent to a large group you can safely ignore it. However, a great way to build up your network and expand your influence is to respond to those broad requests. Just be thoughtful about which ones you pick, for the sake of your sanity and the health of your current project.
- If the request came from your employees and doesn't align with priorities, you should say no and use the opportunity to restate and reinforce priorities. If the team or team member is particularly passionate about an isolated request, remind them that as long as they meet their commitments and don't impact the rest of the team, they can use

their free time as they choose. However, no one will talk glowingly about all the work they did while they failed to do their jobs.

> **Eric Aside** Sometimes side projects can lead to key advances, even when they aren't nicely aligned with priorities. By saying, "As long as you still meet your commitments," you aren't saying "Don't do it." You are saying, "Be sure to still meet your commitments." This is part of the reason that slack time is critical (the other part being time to think—thinking can be beneficial). Ensure that your schedules are always reasonable so they allow for some slack.

- If the request contradicts corporate, division, or team policies, or your personal principles, you should say no and use the opportunity to restate and reinforce those principles. It doesn't matter who is asking—your boss, your general manager, or your vice president. If you can't uphold the principles we, as a company, or you, as an individual, hold dear, then your work and our work mean nothing. We can compromise on a bug or a feature from time to time, but when we lose our principles we lose our soul.

I live to serve

Life is demanding. It's hard to meet all those demands. You have to prioritize. You have to find balance. A constant stream of requests flows through your door, your instant messenger, your wall, and your inbox. While you can't respond to all of them right away, it's important to stay positive and welcoming of what life brings you.

New requests need to be clearly understood, appropriately dispatched, and thoughtfully prioritized against old requests. When you do manage requests quickly, transparently, and responsibly you are called a professional. When you do so with integrity around shared and personal principles, you are called a respected professional. Be a respected professional.

August 1, 2010: "I messed up"

Ever make a bad mistake? One that makes you feel like there's a hollow in your chest—you know you've messed up badly? Maybe you were even trying to do the right thing, but it just ended up wrong unintentionally. This happens to me regularly. It recently happened to a friend of mine—I feel for him.

The worst part of making a bad mistake is the panic that ensues. Your stress level skyrockets. You desperately seek a way to make it better. You can't sleep. You feel guilty and responsible. The panic can last for days. Or you could be indifferent and dismissive because you are pond scum.

Make it right

For those of us with a soul, who care about rectifying our mistakes, the thing we want to know most is "How do I make this right?" I'm glad you asked. Here's what you do, in order.

1. Take responsibility—you made the mistake, you need to admit it.
2. Deeply understand the fallout—don't make it worse by fixing the wrong problem.
3. Invite help to repair the damage—acknowledge the problem isn't yours alone.
4. Ensure it doesn't happen again—uncover and then talk about the way forward.

This can be a tough time to stay calm and in control of your emotions. However, staying calm is essential to rectify the situation. You made a big mess—there are no shortcuts to cleaning it up. Let's talk through each step.

Eric Aside Whatever you did or said wrong is done, so don't bother trying to take it back. E-mail recall only attracts attention to your mail. Hiding mistakes only makes them far worse. Be an adult and a professional—own your mistakes.

Take responsibility

You've made a mistake, proving once again that you are human. Being human is a reason, but not an excuse. You really did mess up. The first thing to do is fess up.

Don't blame others, even if you think they are more at fault. Have some integrity. You acted and others suffered. Admit it plainly. "I made a mistake" is all you need to say.

For a mistake at work, saying "I'm so sorry" and "Please forgive me" can be unnecessary and even legally damaging in particularly sensitive circumstances. Why is saying you're sorry unnecessary in some work situations?

- You aren't sorry in a legal sense—claiming responsibility for everything that happened. You regret making the mistake, but it wasn't malicious.
- You want to inspire confidence in your ability to handle the problems you've caused—not sound weak and helpless.
- You want to focus on the future, when the problems are corrected and don't happen again—not dwell on the past.

Eric Aside Of course, in personal situations it's often important to say you're sorry. However, this is not your personal life. This is business. The best forgiveness comes when you've competently and confidently acknowledged the issue and rectified it.

There's no need to belabor the point. Simply say "I made a mistake" and move forward.

Deeply understand the fallout

The biggest blunder people make after a mistake is suggesting or implementing a quick solution. Take a breath and reconsider. You never would have made the mistake in the first place if you fully understood the problem. Don't pretend that botching something makes you an expert.

You need to deeply understand the trouble you caused.

- Who was impacted?
- Did your mistake impact different people in different ways?
- What was the nature of the trouble (personal, schedule, resource, or something else)?
- What solution do people really desire?
- How can you help, if at all?
- How can this kind of mistake be avoided going forward?

Only after listening carefully to everyone impacted, asking questions, and truly understanding what happened can you possibly know how to make it right.

Invite help to repair the damage

The second biggest blunder people make after a mistake is fixing it alone, out of guilt, hubris, or desire for pity. "I made the mistake—I'll suffer the cost of fixing it" is the rationale. Get over yourself and quit wallowing. Sure, the mistake was yours, but the problem belongs to everyone involved.

Once you know what needs to happen to rectify the situation, ask for help. If you've acted with integrity and listened with care, people will be happy to assist you. Repairing the damage together will rebuild relationships and ensure the solution meets everyone's needs.

Please note the word "relationships." That's what this is all about. When you make a mistake, the lasting harm is damaged relationships between you and your partners and customers. Trust is what you are working hardest to recover.

By asking for help, you aren't trying to avoid responsibility or work. You still own the brunt of the effort and are accountable to see the problems resolved. However, to bring the event to a conclusion that satisfies your partners, they need to be engaged and will want to be included.

Of course, your partners may suggest solutions you don't like—all the more reason to deeply understand the fallout. That analysis will either help you successfully convince your partners to apply an alternate solution or help you accept the solution they desire. After all, you are in no position to dictate to those you've harmed.

The stronger your relationships were before the mistake, the better you will work through the situation together, and the stronger your relationships will be when the trouble is long forgotten. Relationships are everything.

Eric Aside If you run into trouble, contact Human Resources (HR) or Legal and Corporate Affairs (LCA) as needed. Remember, solving things yourself doesn't make you a hero or martyr, it makes you an idiot.

If someone else runs into trouble, forgive mistakes and help your peers recover. The relationships you support may be the relationships you need in the future.

Ensure it doesn't happen again

Once you've taken responsibility, understood the issues, and worked toward a broad solution, the only thing left is to avoid any chance of recurrence. People understand the occasional error. It's when an error is repeated that people question your true intentions and the value you put on your relationships.

As I mentioned earlier, you must deeply understand how this kind of mistake can be avoided in the future. Then you must clearly articulate that intention. There are two magic words you should use: "going forward."

Eric Aside To deeply understand how this kind of mistake can be avoided in the future, you must do a thorough root-cause analysis. I give some examples of using the five whys for root-cause analysis in "To tell the truth" in Chapter 1.

"Going forward I will confer with our stakeholders before finalizing a decision." "Going forward I will run the entire automated test suite before checking in a global change." "Going forward I will ask others before taking the last donut."

Going forward is far better than looking backward. Forget the blame. Stop the whining. Move toward the future. Everyone will appreciate it.

Eric Aside As I mentioned in "Your World. Easier" (see Chapter 3), people tend to repeat certain mistakes, which is why checklists are so handy. Keep track of your personal patterns, and put procedures in place to prevent problems.

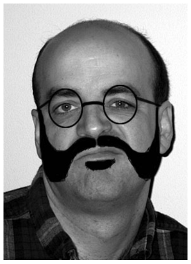
All better

Sometimes people simply need to hear you acknowledge a mistake. They’ll fix the problem or be happy to help. They only want to know that you realize your error, appreciate the problem you caused, and know how to avoid it going forward.

Sometimes there is no way to repair the tangible damage, or the damage lingers for years. However, you can still repair relationships by taking responsibility and ensuring the ruckus is not repeated.

Why are people so forgiving? Because everyone makes mistakes. We’ve all experienced the horror of realizing we’ve messed up. We are here for each other. Trust others to understand. Work hard to understand them. One day, this too shall pass.

March 1, 2011: “You’re no bargain either”



“Can I talk to you about Bozo? He gets on people’s nerves. His communication style causes trouble. He’s bringing the whole team down. He’s a freaking clown.” If you’re a manager, you’ve probably heard this before. Every team seems to have its share of Bozos. What do you do about Bozo? Discipline him? Move him to another team? Fire him? No, don’t be a fool.

Bozo needs to reconsider his behavior—no doubt. If you’re Bozo’s manager, you need to evaluate the situation and take appropriate action. But make no mistake, the problem isn’t just Bozo. The problem is with the whole team.

Let me make it perfectly clear. You are a Bozo. We are all Bozos. No one is perfect. Nobody always says the right thing and behaves the right way. Even if you did, someone would take offense.

Eric Aside Bozo the Clown was the host of a famous children’s show when I was young and dinosaurs ruled the earth.

The good, the bad, and the ugly

“Sure, but not all Bozos are created equal. The Bozo on my team is completely useless—a disaster. He’s got to go.” Really? Why was he hired? What is his review history? Has he been a Bozo from the start?

Sometimes we do make bad hiring decisions and need to send our clowns to a new circus. I designate such truly troublesome teammates as “negative headcount.” If you were to remove

Bozo from his team and not replace him, how would his team perform long-term with one less person? If you believe productivity would be higher and stay higher because Bozo just adds work to everyone else, then Bozo is negative headcount—he subtracts instead of adds to his team.

Most of the time, however, Bozo is a good employee who's made some mistakes. Unfortunately, his coworkers aren't sufficiently patient, understanding, or accommodating when dealing with him.

Dynamics of Software Development is one of my favorite books on its subject. It's written by Jim McCarthy, a former Microsoft employee. Jim's "Don't flip the Bozo bit" essay stresses the importance of giving people a chance. The key is understanding the root cause of the trouble and addressing it with Bozo and the team.

Take all of me

Remember, everyone is a Bozo. We are all package deals. You can't take the benefits of people without also taking their faults. You can fight this notion at your peril, or you can embrace it and accept people as they are.

Why do people "flip the Bozo bit" on others, instead of accepting them as they are? Because it's so easy and natural to project your opinion of people onto them, instead of seeing them as complete individuals. You believe a coworker, a lover, or a celebrity is wonderful, and you see that person as wonderful. That is, until she disappoints you. Then the bit is flipped and she is intolerable.

Stop projecting. Take a look in the mirror. You aren't wonderful and neither is anyone else. You are a real, whole, human being—a marvelous, complex package of positives and problems. So is everybody else. Accept it, embrace it, and move on.

I'll accommodate you

How do you deal with all of life's imperfections? You make accommodations.

Certainly, you should talk candidly and courteously to your teammates one-on-one about their weaknesses. In response, they should try to improve or at least help you both adapt. Remember, you can't fix everything about everybody, as hard as you might try. Instead, know yourself and know your coworkers—adjust accordingly.

I've had inventive teammates who were flakey, brilliant teammates who were arrogant, and insanely productive teammates who were uptight. I could have admonished the flakiness but then lost the creativity. I could have browbeaten the arrogance but then lost the brilliance. I could have completely unwound the uptightness but then lost the productivity. Instead, I provided structure for the inventive flakey folks, tolerance and coaching for the brilliant arrogant folks, and safe places to work and vent for the productive uptight folks.

For myself, I can't keep any details in my head for long, so I write myself e-mails and tell others to write me e-mails. Everyone has a way of working and dealing with his weaknesses. Ask and learn what works well for your teammates and your management. Then get over yourself and accommodate.

Eric Aside As a manager, it's important to recognize when the best accommodation for an employee is to have the employee switch teams. Sometimes you've got a good person who fits poorly with the team. That's not very common, but it happens. Instead of a diversity of ideas, you get a clash of styles. After understanding the root cause, encourage your employee to seek a new role. Help your employee find and attain a happier and more successful future.

We've gotta play to your strengths

Why should others compromise and make accommodations for your weaknesses? Let's say you corrected all your problems, as if that were possible. What would the result be? You'd be ordinary in every area, with no improvement to the strengths that truly differentiate you. You'd certainly get along better with everyone, and that's quite valuable. However, that's not why you were chosen for your role. Accommodate and defuse your weaknesses? Absolutely! Focus on weaknesses while starving your strengths? Never!

Your value as an individual rests in your unique strengths. Those are the areas you want to develop and deploy the most. To the extent that your weaknesses inhibit exploiting your strengths, you must correct or accommodate those weaknesses. However, remember who you are and the core of your success. Adapt, and then accelerate your growth by doing what you do best.

Warts and all

Everyone can be a Bozo at times to people in his work life and home life. Give yourself a break. Accept your faults. Do what you can about them, and accommodate the rest. Help your coworkers, your friends, and your family to do the same.

When talking honestly and respectfully to teammates one-on-one about their shortcomings isn't enough, make accommodations. Tolerate their weaknesses, utilize their strengths, and they'll likely do the same for you. Yeah, you could argue and complain instead, and sometimes that's cathartic, but it's rarely beneficial to anyone.

Swallowing your pride and getting along is healthy for you, your team, and your loved ones. There are always areas where you can't compromise, but typically they are few. Your life will be less stressful, your relationships will be stronger, and you will be happier and more successful if you embrace others as the whole human beings they are.

Chapter 9

Being a Manager, and Yet Not Evil Incarnate

In this chapter:

Part of Microsoft culture is that you don't complain about something unless you've got a constructive suggestion for improvement. Well, I complain regularly about management. What's worse, I've been a manager at the company for 8 of my 12 years. Do I have any constructive suggestions for manager improvement? Funny you should ask.

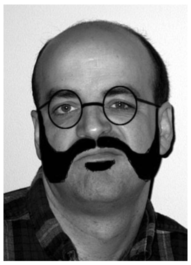
Today there is considerable support for new managers, but when I first became a lead my experiences with prior managers were my only preparation. I did okay, but I learned a great deal on the job and from mentors. After five years, I joined the training organization for Microsoft engineers. My top priority was providing new leads and managers the kind of effective and concrete help I wish I had received. Much of I. M. Wright's writing on management came directly from the materials I created for new managers.

In this chapter, I. M. Wright describes how to manage effectively without becoming demonic. The first column instructs managers on the proper use of metrics and the attributes of a great engineer. The second covers interviewing and recruiting. The third column takes you through the sensitive business of managing poor performance. The fourth deals with retention and turnover. The fifth column *divulges the minimum you need to be a good manager, plus how to go from good to great. The sixth one divulges the secret to transforming a dysfunctional team into a healthy and productive one. The seventh provides step-by-step instructions on change management. The eighth provides the keys to a good recognition program. The ninth reveals why trying to make perfect hires keeps you from making great hires. The tenth rails against unfocused managers. The eleventh discusses the purpose of one-on-ones and morale events and how to make them effective. The last column tells managers how to change the culture of their teams.*

Frankly, I never wanted to be a manager. For 17 years I had been a successful individual contributor and architect. I became a manager because I had my own product ideas and wanted to learn how to run a business. To my surprise, I found managing people to be more engaging and gratifying than programming. Perhaps part of that was the result of being a dad—raising children is similar in ways to growing a team. But mostly it's because compared to people, computers are predictable and unresponsive. Oh sure, computers can surprise and delight you, but not to the degree people can. I still love programming, but I find serving people far more rewarding.

—Eric

February 1, 2003: "More than a number—Productivity"



Is it just me, or have aliens taken over the brains of our managers, convincing them that a number can accurately represent the quality of a product or the value of a developer? Exactly how much data entry, analysis, and forecasting crud must we endure just to placate misguided managers enough to leave us alone so that we can do our jobs? Am I the only one with a deep visceral sense that if we actually were allowed to focus on coding we'd get a lot more done?

Yet the trend is to "measure" more and more of what we create and how we create it and to use these measures to determine the goodness of our products—and of our people. As if disengaging our brains were advantageous. Do you realize how many otherwise intelligent managers could easily second-guess their own best judgment about a person and rely on a "coding success metric" to rate the members of their teams?

Eric Aside I was asked by the *Interface* editors to write a piece about metrics. I don't think this is what they had in mind, but deadlines are deadlines and I provided the word count they requested.

Careful what you wish for

The *Interface* article “Measuring developer productivity” nicely balances information about the various metrics used to measure devs against the weaknesses of using measures at all. Primary among these weaknesses is that metrics are easily “gamed.” Managers will basically get what they ask for. If more lines make the metric better, they’ll get more lines. If fewer check-ins makes the metric better, they’ll get fewer check-ins. Not because the code is better or the developer is better, but because it makes the measure look better.

As long as people believe that they are being judged by a number, they will do whatever is necessary to ensure that the number makes them look good, regardless of the original intent of the measurement system. After all, why go out of your way to do the right thing if it doesn’t improve your standing?

Are all measures useless? Not if they are used to track team and product progress against well-defined and accepted goals—like performance, regression rates, find and fix rates, and days to customer problem resolution. Measures can help drive teams forward and give objective meaning to achievements. However, they never take the place of good judgment.

Eric Aside I’ve since learned a great deal more about constructive metrics. You want them to measure desirable outcomes—ideally, team-based desirable outcomes. Outcomes focus on what instead of how, which provides freedom to improve. Team metrics drive shared purpose instead of competitive dysfunction. “Lines of code” is rarely a desirable outcome. However, “shortening the time to produce a high-quality feature from start to finish” is a desirable outcome, and it depends on the whole team, not an individual. I dive deeply into this topic in Chapter 2 in the column, “How do you measure yourself?”

Rather than attempting to place a number on individual developer productivity, managers should be answering the question, “What makes a good developer?” If it were as simple as assigning a number, theorem-proving machines would have replaced all of us a long time ago.

Playing a role

The key to measuring the value of developers is to think about development teams, not just individuals. Different people bring different strengths to a team. Consider the role that developers play on your team when judging their contributions. The best developers to have are not always the ones who code the best or fastest.

You don’t want a team full of architects any more than you want a team full of leads or a team full of code grinders. Each team needs a balance of talents to be most effective and productive.

The makings of a great dev

But enough skirting around the issue of "measuring a dev." Here are what I consider to be the defining characteristics of great developers:

- **They know what they are doing.** When you ask great developers why any given line or variable is there, they have a reason. Sometimes the reason isn't great ("I took that code from another place, and it used that construct"). But the reason is never, "Aw, I don't know, it seems to work."
- **They don't believe in magic.** This is a corollary to knowing what they are doing. Great developers don't feel comfortable with black-box APIs, components, or algorithms. They need to know how the code works so they don't get burned by a false assumption or "leaky" abstraction (like a string class with simple concatenation that hides allocation failures or O(n2) time to execute).

> **Eric Aside** Kudos to one of my favorite "Joel on Software" columns, "The Law of Leaky Abstractions," which you can find at *http://www.joelonsoftware.com/articles/LeakyAbstractions.htm*.

- **They understand their customers and business.** I talk about this in detail in "Life isn't fair—The review curve," which appears in Chapter 7. Great developers know what really matters, and they can prioritize and make proper tradeoffs.
- **They put customers and the team before themselves.** No task is below a great dev; no customer is unimportant.
- **They have uncompromising morals and ethics.** Although individual preferences may vary, great devs care about how they accomplish their work and how they act toward others. Whether in the algorithms they choose or the e-mail they write, they set the bar high for themselves and will not waver from their core values.
- **They have excellent people and communication skills.** Although not many developers would make good game show hosts, great developers work well with others, respect others, and communicate clearly, effectively, and appropriately. They don't choose to bully or intimidate (although they could), but instead collaborate. (More on this in "My way or the highway—Negotiation," which appears in Chapter 8.)
- **They have a wide, supportive network.** Great devs recognize greatness in others and are drawn to each other. They quickly develop a network of contacts that support one another and that allows them to be far more effective than any single individual. (For more on building your personal network, read "Get yourself connected" in Chapter 7.)

Other specific aspects of the general characteristics I've listed include focusing on quality, mentoring others, and displaying exceptional design skills.

None of these characteristics that define a great developer can be simply measured.

You be the judge

When push comes to shove, you as a manager must judge your team as fairly as you can in the roles that each member plays. Having examples from other teams helps give you perspective on your own developers; this is why calibration meetings are so valuable, instructive, and worth the agony.

But remember that no calibration or rating can hope to represent an individual. People are far too complex, and even the most objective metrics are distorted by perspective. Knowing and valuing people as real human beings is the key to unlocking their full potential.

September 1, 2004: "Out of the interview loop"

Recruiting is like a huge vacuum sucking up all my time on campus. But quickly hiring a quality candidate is worth every minute. Luckily, I'm not dependent on anyone for most of the recruiting process. This keeps my candidates rolling through. That is, until the interview loop.

Man, if I could only skip the interview loop, I'd have it made. No scheduling hassles; no multi-week delays for a time slot; no horrible interview questions; no disappearing feedback; no "borderline" hires; no vague, gut-based garbage criteria; no freight train of duplicate thinking; and no last-minute cancellations.

But you can't skip the interview loop—ever. If Bill Gates wanted a job on my team, I'd put him through a loop. No offense. Just put him through the loop and make sure he's going to be a successful addition to the team—then brace myself if he's a no hire.

Blaming the help

Some of you hiring managers may take exception to my statement that I'm not dependent on anyone until the loop. You may say, "What about my recruiter? My recruiter doesn't give me the time of day. My recruiter hasn't sent me a resume in months. My recruiter is the bottleneck." Man, am I glad that I'm competing against you for hires. If it weren't for you lazy, incompetent fools, I'd have a heck of a time stealing all your strong candidates.

Your recruiter is your partner, your friend, and your resource—not your servant. Recruiters have WAY too many positions to fill. Unless your VP thinks your open position is critical, your recruiter cannot do all your grunt work. So get over yourself, get over to your recruiter's office, and get all over their stack of resumes. Otherwise, move over while I get my positions filled.

Eric Aside Today all the resumes are online, of course. (They pretty much were back then too.) However, the point remains that if you don't find your own candidates, your candidates will find some other job.

Ninety percent preparation

Now, back to the interview loop. (I'll have to save general thoughts about recruiting for a future column.) So many hiring managers get the interview loop process wrong that I'm just going to lay it out for you. A successful loop is 90% in the preparation, and the rest lies with your As Appropriate interviewer. The preparation falls into three steps:

- Prepping the interviewers.
- Prepping the recruiter.
- Prepping the interviewers again.

Anyone who can interview should prepare in advance by

- Taking interviewer training.
- Developing strong interview questions.

Without interview training, your junior people won't learn the proper technique and your senior people won't learn what bad habits they've acquired. Without strong interview questions, you might as well hire Jason Voorhees.

That is the question

Ah, the interview question. What makes a strong interview question? After many years of interviewing, I've decided there are only two types of worthwhile interview questions:

- Questions that expose personality traits.
- Questions that demonstrate how the candidate will perform on the job.

Brain teasers are worthless, background questions are a yawn, and "How would you do this job" questions are mindless regurgitation. Rarely do these types of questions directly expose personality traits or truly demonstrate performance. That is, unless you use the very best follow-up interview question of them all, "Why?" Not once or twice, but repeatedly until you get to that key personality trait or performance characteristic.

Even better, start with a "why" question. For instance, "Why do you want this job? Why are you leaving your old job? Why are you still working for Microsoft? Why do you want to work for Microsoft?" Again, these questions are useless if you stop asking "why" after the first response.

You are looking for how this person aligns with our key competencies, particularly passion, follow-through, flexibility, integrity, and professionalism. Anyone can show these on the surface. Keep asking "why" to get below the surface and find real evidence one way or the other.

The whiteboard compiler

Aside from "why" questions, there are coding questions (or other similar technical problem-solving questions) that try to uncover how candidates will perform on the job. These can be strong interview questions. The issue is coming up with the right questions.

Rather than bore you with countless examples of bad, overused, web-publicized questions, I've got a step-by-step guide for coming up with your own great new questions:

1. Choose two or three real problems that you or your team have worked on over the past 18 months. The solutions should fit on a single whiteboard and involve at least three different variables. This ensures that you are selecting challenges with short and nontrivial solutions. In general they are small functions, pieces of a design problem, or particular test cases.
2. Break down each problem into a simple core issue, and use that as the first question. As a candidate builds confidence, add more complications that increase the difficulty. For instance, have the candidates look for more optimal solutions, introduce new cases, or ask for a more robust "production-quality" solution.
3. Call out areas that are gray in their analysis, and push the candidates for answers to see how they respond to questions outside of their safe zones.
4. Be prepared for multiple problem solutions, and point out real issues from your real problems.

Discard problems that are over two or three years old. You always encounter new problems, so rotate old problems out. It makes interviewing more fun for you and more relevant for the candidates. It's also much harder for candidates to read the answers on the Internet.

Eric Aside There is a huge tendency to hold onto old problems because it's easier and you've already learned how to gauge candidates' solutions. Get over it. Recent problems are better.

The goal is to see how each candidate approaches and solves the problems, not to get the right solution. There are a wide variety of things that could keep great candidates from finding a solution during the few minutes that they have with you. Instead, look for how they display core competencies while they are problem solving, such as the following:

- Did they identify if a strategy isn't working?
- Are they asking you questions to help them get on the right path, and do they listen to your hints?
- Did they analyze their process and results?
- Did they apply multiple strategies?

Be present in mind and soul when each candidate is at the whiteboard. You aren't looking for the answer; you are looking for how they get there.

Much of this information can be found in the Interviewer Toolkit, a great resource that HR put together with the help of a number of dev managers, including myself. You can also share and critique questions with your friends and dev team, but be careful not to duplicate their questions or soon the solutions will be published on the web.

Eric Aside These days I often need to assess competencies that don't lend themselves to white-board problem solving, so I use role playing instead. The basic premise is simple: if you want to evaluate how well someone can code, have them code; if you want to evaluate someone's confidence in their decision making, question their decisions.

Online Materials Interview Role Playing, a how-to guide (InterviewRolePlaying.doc)

Prepping the recruiter

After your interviewers are prepared with potential questions for candidates, you must prepare your recruiter. Send your recruiter the job description (including title and level information) and a long list of prepared interviewers. The longer the list, the easier it is to schedule interviews.

If you are just staffing up, borrow interviewers from your peers' teams and spread around the interviews to avoid creating a high burden on any one individual. Then be prepared to pay back the favor. Regardless of how you find the potential interviewers, let your recruiter choose from many names.

Present your interviewers like a menu: For the first interview, choose from one of these fine selections; for the second interview, indulge in one of these excellent choices. And so on. You can repeat names and be creative, just make it as easy as possible to schedule and you'll get more loops scheduled faster.

Prepping the interviewers (again)

Finally, the day before the interview comes and, with it, the interview feedback instructions thread. Reply to this thread right away with your instructions for the interview loop. Tell your interviewers about the position, what you are looking for in this particular candidate, and how you want the interviews conducted.

Each candidate can bring different things to the position. Each candidate may interest you in different ways. You must describe this to the people in the interview loop as well as describe your sense of the candidates' strengths and weaknesses.

Next, tell the interviewers what type of competencies you want each of them to focus on. Each interviewer should have a clear mission with as little overlap as possible. Leave room for a late interviewer to follow up on issues raised earlier in the day, but make sure every interviewer has a role. This prevents repeat questions, wasted time, and painful oversights.

A gentle reminder

Finally, remind the interviewers of the following:

- Speak briefly and privately to the next interviewer before that interview starts. Talk about the candidate and your hire/no-hire decision. Let the next interviewer know what kinds of questions you asked and what kinds still need to be asked.

  > **Eric Aside** Given that interviewers each have a clear mission with little overlap, you may wonder why it's necessary to tell the next interviewer your impressions or what you asked. It's because life is unpredictable and humans are human. Maybe your interviewer changed things around. Maybe the interview got off on a tangent. Maybe some interesting character trait arose that deserves special attention. Who knows? Being flexible is a good thing.

- Write and send feedback promptly after the interview.
  - Start with your hire/no-hire decision.
  - Follow that with a short summary of your impressions, both gut and substantive.
  - Add concrete examples from your interview that back up each of your impressions. Quotes from the candidate are especially useful.
  - Include what questions you asked and what questions still need to be asked.
  - Conclude as you wish.
- Do not use the word "borderline" in your hire/no-hire decision; make your choice and then defend it. By default, a borderline hire is a no-hire.
- Do not be afraid or embarrassed to disagree with the prior interviewers. Say what you really feel, what you really saw and heard.

The last puzzle piece

With this preparation, the people on your interview loop should be ready to give each candidate a strong interview and reveal real insight into how successful the candidate could be in your open position.

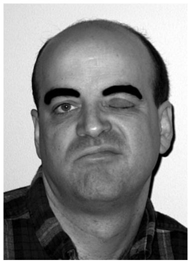
The last piece of the puzzle is your As Appropriate interviewer, the last interviewer in the loop. Like the rest of the loop, you should have a few As Appropriate interviewers available on your interview loop menu. The As Appropriate should be

- At least as familiar with your position and expectations as your recruiter. Ideally, you should discuss the position with the As Appropriate personally.
- Engaged in the interview process all day, gathering late feedback, correcting and illuminating poor feedback, and focusing on trouble areas.
- Prepared to sell your position and the team if the candidate is a strong hire.

Eric Aside The last interviewer at Microsoft is called the "As Appropriate" because he or she is a senior person and will do the last interview only if it's appropriate—that is, if the candidate shows real promise.

With a good set of interviewers, strong preparation on all fronts, and an aggressive recruiting effort, your interview loops will become your strength instead of your weakness, your propellant instead of your weight. You'll get your positions filled quickly, with great people, and be back at full strength, ready to deliver.

November 1, 2004: "The toughest job—Poor performers"



Reviews are over, and you may have pondered whether or not you'd be better off on a different team. You know, the kind that gives out more 3.5s and 4.0s. The kind where you'd be surrounded by lame wannabes and get a promotion faster. You've probably decided against that kind of move—a wise decision. However, you may be wondering why this inequity exists. Where did those lame wannabes come from? Why are they here? What's being done about it?

Eric Aside Actually, lame teams get exposed within a year or two, but it would be better for everyone involved if they never got started. That's the point of this column.

Well, take a look in the mirror. Have you allowed poor performers to slide by, thus giving your group easy annual 3.0 targets? Have you let entry-level employees stay entry level for years? Have you passed mediocre employees on to another manager or team rather than deal with them? Guess what? You are the problem.

Eric Aside In Microsoft's old rating system, ratings of 2.5 and 3.0 were undesirable. Ratings of 4.0 and 4.5 were highly desirable. A 3.5 rating was readily accepted and the most common.

What did you expect?

The poor performers aren't the problem. As far as they can tell, nothing is wrong. Sure, they aren't excelling, but they are getting by, and that seems to be enough. You haven't driven higher expectations for their performance. As a result, the poor performers stay. They find groups where they can be more comfortable. They hire mediocre folks like themselves. Finally, they drag down their orgs, then scatter like rats from a sinking ship.

But it's not their fault; it's your fault. You must expect more from them. I know it is hard. Poor performers aren't necessarily mean and nasty. They are often kind and thoughtful. They have families and obligations. They care about doing a good job, and they try to do the right thing. Telling them that caring and trying aren't good enough isn't easy.

Get over it and get over yourself. Going easy on mediocre employees is disrespectful and distasteful. You aren't doing them, yourself, or the company any favors. In fact, you are cruelly and selfishly harming all three.

Why? Because you are setting up your underperformers for failure. You've managed to create a situation where every workday your employees wake up, get dressed, and head into a job in which they will fail. Do you know what that feels like? You owe them the decency of explaining where they stand and what they can do to improve.

Eric Aside I'm being overly harsh here to make a point. If managers said nothing, that would be setting up poor employees for failure; however, managers do typically discuss performance with employees. Unfortunately, they often aren't as clear and firm as they should be, because they don't want to seem cold-hearted or cruel. Actually, being clear and firm is the kindest, most constructive thing managers can do.

Bite the bullet

So, say you've got employees who aren't meeting your expectations in one or more areas. What do you do? Simple. In your weekly one-on-ones, talk to them about your expectations, where they are falling short, and what meeting your expectations would look like. Tell them not to worry about writing it down, you'll send an e-mail.

Why send your expectations in an e-mail? It's all for the lawyers, right? Wrong! You write down your expectations so that they are clear and unambiguous.

One of the biggest problems with performance issues is inherent miscommunication. If your employees were clear on expectations, there wouldn't be a problem. When they are falling short, your employees must be crystal clear on why and what they can do to improve. A short e-mail will do; include bullets on each of your expectations, where they are lacking, and what success would be.

Seeking professional help

But what if you don't see improvement? What if you feel a 2.5 may be in progress? Contact your HR generalist immediately. Your generalist will work with you to ensure that your employees are getting the right information and the right help if they need it.

Sometimes performance issues are caused by personal matters. Depending on the situation, Microsoft may have an obligation to accommodate employees through such times. Your HR generalist will know all the options and can properly get your employees the help they deserve. Do not play psychologist, doctor, or lawyer for your employees. Leave those matters for your generalist to refer employees to qualified agents.

Failure is not an option

If the performance issues persist, your generalist can help you find the best course of action. While there are cases of re-leveling or changing roles, the more common situation comes down to presenting employees with three choices:

- Voluntarily leave the company
- Improve
- Involuntarily leave the company

Microsoft is very good about helping employees make the transition, which is why having your generalist involved is so beneficial. Often when presented with clear options and failing expectations, employees will be relieved to leave the company and find a new opportunity where they have a better chance of success. This is the second best result, and it happens more often than you might think.

Eric Aside Not every employee with performance issues gets the same choices. HR can help you tailor the proper response to each unique situation. Here I'm describing a common set of choices.

The goal is success

The best result is when your employees improve their performance enough to meet or exceed your expectations. You get back great employees. You don't need to fire anyone. You don't need to spend months hiring someone new. You don't need to bring new people up to speed. Your employees keep their jobs and their esteem, and they become successful. It's a huge win for everyone.

So, if your poor performers choose to improve, you must believe that they can and be part of their success. Even if you are sure that this person has no chance of turning his career around, you must trust that he will somehow overcome the obstacles.

Why? Because it's better if he succeeds, and because he knows if you doubt him. People can tell when you don't believe in them, and they assume it means that they don't have a chance to succeed at Microsoft. Remember, if you don't believe that your employees can change, you are setting them up for failure. That can have tragic consequences.

Ask and you shall receive

To set up your employees for success, you must go back to setting clear expectations for solid performance. Your expectations must be in writing to minimize misinterpretation and misunderstanding. Review these expectations weekly, and discuss in writing and in person where there is improvement and what remains to be done.

Often poor performers will show an initial spike in improvement. Managers get excited about this and tell the employees that they are pleased, only to have the employees top out or even regress. The problem is that the employees get the sense that they are exceeding expectations, when in fact they are doing better but still falling short.

You must be supportive, but focus on success by saying, "Hey, it is so great to see your significant improvement. I really appreciate your efforts. By continuing this trend and improving in these other areas, you will begin to meet the expectations of your assignment and be on the road to success." This reassures an employee that you notice and care, while still setting a clear bar.

Often a poor performer who doesn't have what it takes to turn things around will realize her shortcomings after she puts in her best effort and still falls short. If you support their success, employees can leave voluntarily on good terms knowing that you treated them fairly and with respect. You gave them a real chance.

You can't always get what you want

In cases where poor performers are too stubborn or prideful to admit that they can't meet your expectations, you will have all the proper documentation needed for an involuntary resignation (firing). Your HR generalist will help you and your employees through this difficult step.

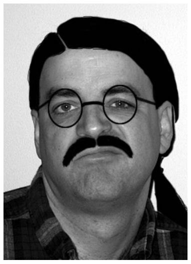
If you do need to fire an employee, you'll have to send out the awkward e-mail to the group, "[Employee] is no longer with Microsoft and will be pursuing other opportunities." The most difficult aspect of this is that you can't say anything about why it happened. You must protect your former employee's privacy, even if that employee chooses to tell people a different story.

It is common for members of your group to wonder what happened. They'll ask you why an employee left. Although you are not permitted to tell them anything about the private situation of your former employee, you can answer their real question. When people ask about someone else's situation, they almost always are truly concerned about their own job security. That's selfish perhaps, but also quite natural.

Use the opportunity to give them feedback that will help them perform better and to reassure them about their own standing. You can also reassure them that Microsoft has strong expectations of employees because the company knows that we will only succeed when we have extraordinary people performing at their best in a healthy and supportive environment. That is a message we all can be proud of.

Eric Aside What do you do if you meet a former employee at the supermarket or parking lot? Won't that be awkward? No, it really isn't. I've had my share of poor performers. I had to fire a few of them, and I do occasionally run into these former employees. It's great to see them. Because we were constructive and truly shared the same goal—setting up the employees for success—the employees are typically quite happy to see me. We talk about their career progress, and usually it is going better than before. That's the whole point.

September 1, 2005: "Go with the flow—Retention and turnover"



Review season is here. As entertaining as that can be for managers and employees alike, it's just a primer for what follows: musical product groups. The music starts when review numbers are released. A whole bunch of engineers get up out of their office chairs and try to find an available chair in a different group. The music stops when all the interesting positions get filled. The same game gets played at the end of product cycles, but product cycles aren't quite as predictable (different problem).

It's tough on the folks left without chairs. Often they get alienated from their current teams for trying to jump ship. Tough luck? I don't think so. Managers should encourage their people to pursue new opportunities, including people they like. Anything less is selfish, stupid, and shortsighted, not to mention destructive, delusional, and deplorable. When managers make business decisions that put their interests ahead of Microsoft's, they've clearly stopped working for Microsoft. I think Microsoft should stop paying them.

Don't tell me, "Oh, but the project depends on this person. It's necessary for me to alienate him and stifle his personal development. It's for the good of the company." That's a crockload of dung. What you're trying to tell me is that you didn't plan for turnover, and now that it's come, you want to avoid all the work to recover, recruit, re-educate, and reassign people. In other words, you're brain dead and lazy, but you're making up for it by being selfish and self-serving. Only ignorant nimrods are unprepared for turnover.

Eric Aside Okay, I've held back till Chapter 9, but I have to say it, I love rereading these columns. Writing them has always been a cathartic experience. Reviewing them for this book has allowed me to relive the satisfaction of unmitigated rage directed at behavior I truly despise. It's nice in its own twisted way.

I'll just walk the earth

Good managers should expect around 10% turnover a year. Bad managers should expect more, but they probably don't recognize it, and I certainly don't care.

If you're a good manager of a group of 20, you should expect two people to leave your group each year, sometimes more, sometimes less. Even a lead with five reports should expect at least one person to leave every couple of years. People leave for all kinds of reasons, many of which have nothing to do with you or your team: friendships on other teams, new technologies, a change of scenery, and relationships outside work to name a few. You shouldn't take it personally.

Nice dam, huh?

But how should you deal with turnover? Some managers go to extremes to prevent it:

- They blow tons of money on extravagant morale gifts and events, when having more frequent, cheap events would be far better.

  Eric Aside You can read more about running effective morale events later in this chapter in "One to one and many to many."

- They promise larger roles and promotions—promises they don't completely control, promises they can't keep.

- They deny permission to interview for everyone on the team, which poisons morale and makes the team feel like indentured servants.

Eric Aside Microsoft has since changed its rules so that managers can no longer refuse to allow their employees to interview. Only vice presidents still have that privilege.

Trying to prevent turnover is like building a dam to prevent a river from flowing. It works for a short time until the dam breaks or overflows and your team gets washed away in a torrent of transfers. What's worse, such measures lower morale and make your team less attractive to the new members you'll soon need.

Flowing like a river

Instead, the best way to deal with turnover is to expect it and embrace it. How? Think flow, flow, floooooooow.

Think of your team as a river instead of a lake. A lake stagnates. There's no energy or impetus to change. The same is true of groups that stagnate. They cultivate mediocrity and complacency; they abhor risk. A river is always running and changing with lots of great energy. You want a river.

A river depends on the flow of water, and your team depends on the flow of people and information. You can think of the people divided into three groups: new blood, new leaders, and elders ready for a new challenge. Here's how those groups should balance and flow:

- The largest group should be the new blood. Not all of them will become technical or organizational leaders.
- Sometimes you'll have more new leaders than elders, sometimes the reverse, but ideally you should maintain a balance.
- For flow, you want a steady stream of new blood becoming your new leaders, and new leaders becoming elders.
- The key to flow is new blood coming in and elders moving out. For this to work, you WANT your elders to transfer before they clog the stream and disrupt the flow of opportunities for others.

Not all technologies flow at the same rate. Central engines, like the Windows kernel, flow slowly, while web-based services, like MSN Search, flow quickly. You need to adjust for your situation, but even the most conservative technologies do change and flow.

How do you successfully encourage and maintain a healthy flow?

- Keep a constant supply of new people.

- Instill information sharing as a way of life.
- Shape the organization and roles to create growth opportunities.
- Find new challenges for your elders.

Fresh meat

For a constant supply of new people, nothing beats interns and college hires. Obviously, you'll also recruit industry candidates and internal transfers, but interns and college hires should be your primary choice for their fresh perspectives and long-term potential.

Your number of annual college hire slots should be at least 5% of your total staff, counting open positions. So if your team has 20 devs, you want at least one college hire slot, more if your team is increasing headcount. Even in a flat headcount organization there is still at least 5% attrition, so look for young talent to fill openings even if none are currently available. College hires sometimes don't start for nine months; anything can happen over that time, so plan ahead.

Interns are the next best thing to college hires, but they take an extra year to join your team. Therefore, you want as many intern slots as college hire slots. DO NOT plan on shipping interns' code. At best, they should be pair programming shipping code. However, DO NOT give interns menial labor either. Instead, give interns strong mentors (people who'll be your next leads) and exciting projects (buddy them up on the coolest features or incubation work). You want to measure them as future full-time hires and convince them that there's no better job in the world than working for you at Microsoft.

Sharing is caring

When you have new folks on your team, you want them to grow into your new technical and organizational leaders. The only way this happens is through sharing information and knowledge. There is a cornucopia of ways to do this. Here are just a few:

- Keep an online knowledge warehouse of how your group works. It can be a big, versioned Word doc; a SharePoint site; or a wiki—whatever works best for your folks. The key is to make it easily accessible and up to date. The first month's assignment for a new person should be to update the content.

> **Eric Aside** My favorite way to do this now is with a OneNote notebook hosted on the team SharePoint site. It's terrific because it's even easier to edit than a wiki, has wikilike linking ability, has a beautiful offline sync story, and looks great on the web with the new Office Web Apps. The only downside is the lack of automatic wikilike page versioning.

- Use buddy systems for all projects and assignments. The arrangement can be anything from mentoring to assigned reviewer and backup to full-on pair programming. The key is to have no one working alone, no information isolated.
- Get people together regularly, ideally daily, to discuss progress and issues. Nothing encourages sharing of information like regular high-bandwidth communication, even for as little as 15 minutes a day.

Buddy systems are particularly important for growing your new leaders and transitioning your elders. It's never safe for an elder to leave if you lose key knowledge and capability in the process. By constantly sharing information, you release the stress-inducing stranglehold on your elder team members, and you make flow and transition a positive and natural experience.

Room to grow

Just like with repotting plants, you need to give your people room to grow. You can encourage this through how you structure your organization, how you issue your assignments, and how you design and promote growth paths for people to follow.

First think about growth paths. The new career models provide excellent and detailed guidance. How do growth paths apply to your team? You should know every employee's desired growth path and current stage. Then you and your leaders should discuss if those desired growth paths are available for everyone, and if not, how will you adjust?

Often how your group is organized blocks your employees' growth. All your senior people may be on one team and newbies on another. Change it, fix it, rebalance, reshuffle. The longer you leave your org unbalanced, the more trouble you'll cause and risk you'll carry.

Restructuring your organization can create dozens of new opportunities for growth. It's critical to take advantage of them. Give your people assignments and new responsibilities that stretch them out of their comfort zone. Naturally, buddy them up with more experienced partners to reduce the risk and enhance the learning, but don't just have the same people do the same things. Choose the assignments based on desired growth paths, and everyone wins.

I must be traveling

Of course, no one can move up if your most senior people stay put. Unless your group is expanding, the only way to make room for growth is to have your elders transfer out. Luckily, that's exactly what they need. If they've been in your group long enough to reach the senior positions, then the only way for them to keep growing is to take on new challenges elsewhere.

Because you've focused on flow, losing your senior employees is no big deal. They've already shared their knowledge and experience. Their project buddies are already familiar with their work. Now all they need to do is find a good fit elsewhere with you supporting them every step of the way. This kind of loyalty and support will not only be appreciated by your senior people, but will be returned in loyalty and respect by the whole team.

Remember, the whole team watches how you treat your most senior folks. It's an indicator of how you'll treat them some day. Nothing wins over a staff like seeing the elder members being treated fairly and generously; leaving the group with praise, well-wishes, and a great future ahead. The message: "Stay with this team and you'll be well rewarded."

Surrender to the flow

When you fight turnover or let it catch you unprepared, you risk your project and the effectiveness and health of your team. When you embrace turnover, it becomes just a natural consequence of life. No fear, no worries, just healthy flow for an effective team.

What's more, driving for flow of people and information in your team creates growth for Microsoft people and value for Microsoft customers. Less stress, more opportunity, greater flexibility, compounded knowledge, higher morale, and a stronger team. What more could you possibly ask? It's time to surrender to the flow.

December 1, 2005: "I can manage"

What do weddings, travel, and managers have in common? Talk to any adult about these topics and you are sure to hear a horror story. At weddings, it's the drunken guest, bad weather, or untimely faux pas. During travel, it's the lost baggage, disruptive passenger, or transit foul-up. With managers, it's the frigging incompetent, clueless, arrogant, insensitive, conniving, beady-eyed, spineless, self-serving jerk who used to be your boss. Not that I'm bitter or anything.

Skipping the extreme cases, most wedding and travel horror stories can be retold with a smile and shared laughter. The same can't be said for manager horror stories. Sure, everyone will have a good laugh at the stupidity or absurdity of an old boss's actions, but when you look into the eyes of the employee who suffered, there is always simmering contempt. That employee has not forgiven or forgotten what that former manager did.

The gift that keeps on giving

Why do people so easily let go of wedding and travel mishaps, yet clench tightly to manager malfeasance? It's because at the end of weddings, the couples kiss. At the end of a long trip, you finally return home. Bad managers stay. There's no happy ending. Bad managers are there every day, day after day, making one bad move after another.

Even when you finally escape a bad manager's grip, his legacy hangs on with lost time, lost opportunities, and lost results. His comments and actions haunt you. His past callousness compromises your current perceived and real sense of worth. You become distrustful of managers in general. It can take a decade to repair the trouble caused in a year—and that's just for one person.

The damage done to the company is even greater. Rotten managers generate lost productivity, waste and rework, poor quality, blown commitments, and disgruntled employees who quit and complain or stay and sabotage. Rotten managers also generate potential lawsuits, but that's a whole other subject.

Good enough for me

"Hey, give managers a break," some might whine. "Being a good manager is hard." No, no it isn't. Being a great manager is hard. Being a good manager is easy. A good manager only has to focus on two things—two very simple things that anyone can do:

- Ensure her employees are able to work.
- Care about her employees.

That's it. No magic, no motivational videos, no 24-hour days are necessary. A good manager just needs to ensure his employees can work, and he must care about them.

Easy does it

Ask even the most pubescent managers how to ensure that their employees are able to work and they'll say, "Remove roadblocks." It's obvious, isn't it? But ask, "What kind of roadblocks?" and they'll say, "Tracking down dependencies," "Hounding PMs for specs," or "Demanding decent repro steps from testers." So misguided.

Why is a frontal assault on cross-group barricades misguided?

- When managers try to be heroes, they only manage being idiots. They track down dependencies, resulting in rushed, lousy drops, instead of collaborating on BVTs and API designs and having realistic plans and contingencies in the first place. They hound PMs for specs instead of collaborating on all aspects of the design. They demand decent repro steps from testers instead of providing instrumentation that allows for easy debugging of failures from any source. In other words, misguided managers make themselves part of the problem instead of part of the solution.

- Cross-group barriers are hard and time consuming to break. Managers need to give their teams workarounds before undertaking valiant quests.
- Misguided managers allow cross-group roadblocks to distract their attention from more immediate, basic concerns.

Of course, cross-group issues are important to resolve. But there are usually straightforward ways to help with or work around the issues, and more basic problems to address first.

> **Eric Aside** I'd say that one of the top mistakes new managers make is focusing on removing complex roadblocks rather than the simple ones I describe next. The other top mistakes new managers make are continuing to think of themselves as individuals instead of team representatives and not delegating properly. (See "Time enough" in Chapter 8.)

I want to work

So what basic necessities should managers provide to ensure their employees can work? As I said, that's easy. Engineers don't need much, they just need

- A desk with a phone
- Power (light, heat, electricity)
- A computer with a keyboard and display (some don't even need a mouse)
- Network access and privileges
- A healthy environment (safe, relatively quiet, with breathable air)
- Work to accomplish

Who could mess this up? Maybe finding work for people could be complicated, but usually that's not a problem. However, there are plenty of managers who allow employees to go for days without a computer—managers who don't have an office ready for a new team member.

What happened the last time your network went out or the noise got too loud? Did your manager drop everything and do whatever it took to fix the issue? If not, she failed utterly in her role as a manager. Nothing is more basic than ensuring your employees can work.

What about providing a safe environment? Is there any hostility on your team? What is your manager doing about it? This is why antiharassment training is necessary. It's because there is nothing more critical, more essential than ensuring everyone is given an opportunity to work in a safe environment.

I'm not an object

The second simple thing a good manager must do is care about his employees. This doesn't mean warm hugs or greeting cards. You don't even need to like them. You just need to care about your employees, seeing them for what they truly are: fellow human beings.

Again, who could mess this up? How hard can it be? Yet managers commonly think of their employees as resources instead of people. They label their employees as the "good ones" and the "bad ones." They turn their employees into objects instead of seeing them as human beings.

Ever have a manager play favorites? It hurts, doesn't it—even if you are one of the favorites? That's because you're always one false step away from being shunned. A manager who plays favorites is a manager who has stopped seeing her employees as people and started treating them as collectibles.

Taking the time to know and appreciate your employees as real people isn't complicated, but it is a commitment. You need to set aside your own preferences and prejudices and let others know you so you can know them. I realize this is mushy stuff, but good managers respect their employees as real people, they don't treat them as abstract headcount.

By the way, you don't need to say or do anything to show you care. In fact, no amount of words or deeds can convince people you care about them if you really don't. People simply can tell. You can yell at them, praise them, criticize them, and even disappoint them—employees will see through it all and still respect you if you respect them.

Remember, all you need to do to be a good manager is ensure your people can work and treat them as human beings.

Eric Aside It's a necessary and sufficient condition—meaning if you ensure your people can work and you care about them, you are a good manager (really one of the better managers). If you don't ensure they can work or you don't care about them, you are a bad manager regardless of your other herculean efforts.

Good to great

The world and our company would be a better place if all managers were at least good managers. However, Microsoft is a competitive place, so you're probably wondering what makes a great manager.

Many books, journals, and graduate schools have been dedicated to defining great managers. To me it comes down to three aspects:

- **Be a good manager** Don't start by being great; start by being good. The moment you forget the basics is the moment you lose the respect and effectiveness of your employees.

- **Have integrity** This means align your words and actions with your beliefs. If you believe in a strong work ethic while maintaining a clear work-life balance, demonstrate it. If you believe quality comes first, make it first. You set the bar. You define the team.
- **Provide clear goals, priorities, and limits** Clearly communicating your goals, priorities, and limits up, down, and across your organization is essential for highly effective teams and individuals. Goals can take the form of expectations, commitments, or vision statements. Regardless, they show where to go. Priorities show how to get there. Limits provide the safe boundaries within which to travel. Without goals, it's easy to go nowhere. Without priorities, it's easy to get lost. Without limits, it's easy to stumble.

Being a great manager is difficult. You must stand up to the pressure to compromise your beliefs. You must clearly and strongly communicate a consistent message about your vision, what's important, and what's not acceptable. You must do all this while not losing sight of the humanity of your people and the small, but important, things they need to be productive.

I serve

That's all there is to being a good, or even great, manager. It all comes down to service. You are no longer the one doing the work. Instead, you subjugate yourself to the service of your team members so that they can be successful. Management, when done well, is about selflessness. There is no higher calling.

Eric Aside Feel free to share this column with managers you love or loathe. Nothing I've written in the more than five years since encapsulates everything about being a good and great manager more completely and succinctly.

May 1, 2006: "Beyond comparison—Dysfunctional teams"

This column is for leads and managers, but I'll bet you report to one, so feel free to share.

How often do your employees rant about each other? Do they form "secret" alliances? Is the air thick with tension in your team meetings? That must be tough for you; my teams always get along great.

Does your team have lunch together? Are your morale events strained? My team eats together once a week and messes around at a morale event once a month. We laugh almost the whole time.

When your team is stressed, do they fracture? When one of your employees falters, do the others take advantage of the situation? Do they just stand back and enjoy the moment? Or does your team look after each other? My employees always band together and support one another.

If we reported to the same manager, how would you feel about the job you are doing? What would our manager say? Would she tell you to be more like me? Would she throw my team's survey, retention, and productivity numbers in your face? How would you feel, especially at review time?

Trying to pick a fight

Think I'm picking a fight? Darn right I am. If our manager compared my team to yours and shoved the facts in your face, you'd want me dead, or at least bucked off the high horse the boss handed me. You'd probably try to sabotage my success and make me look like a dimwit.

Sound familiar? If your team is dysfunctional, that's what is happening. Members of your team are sabotaging each other and pressing their own agendas to improve their relative standing. Teamwork is for people uninterested in mortal combat.

Who or what is the culprit? Are your people too competitive? No way, I've got highly competitive people on my team who used to be on yours, only now they get along. Is it the review system that pits people against one another? No, I use the same system. Is it the stress level? No, my people shine brightest when they are challenged.

What's the difference? Why do my teams function so well? Want to know? Here's the answer: I never compare my people to each other; I compare them to their own potential and my expectations. That's it. That's the secret.

Go ahead, try to deny it. It can't be that simple, right? Besides, I have to compare them to each other; it's part of reviews, right? Wrong. It is that simple, and reviews don't have to be destructive.

Eric Aside I took a dangerous tact of being directly confrontational with my readers and seeming quite conceited for this column. It was deliberate; I wanted my readers to relive that visceral animosity they felt as children against their rivals. Since I. M. Wright is an arrogant son of a female dog anyway, I figured his reputation wouldn't suffer and trusted people could handle it.

This is not a competition

I wish I could claim to have discovered this secret on my own, but I'm not that smart. I learned it from a parenting book, *Siblings Without Rivalry: How to Help Your Children Live So That You Can Live Too* by Adele Faber and Elaine Mazlish (HarperCollins, 2004). The premise is simple: children want your love and attention, and if you compare them to each other, you’ve declared it a competition. However, it isn’t a competition unless you are a really sick parent. Therefore, don’t compare.

The same goes for employees. They all want the admiration and appreciation of their supervisor. If you compare employees to each other, then you’ve declared it a competition—but it isn’t. Yes, employees compete for rewards division-wide, but at that scale the competition isn’t directly between your team members.

Yes, there are donkeys posing as managers who insist their employees should compete for rewards at the team level. However, that is so far from reasonable, respectful, and rational that I’d like to personally report every case to HR. Donkeys like that shouldn’t be allowed to manage people. They should be relegated to walking in endless circles by adults in clown suits while screaming children pull their hair.

When teammates don’t compete against each other, their best chance to achieve success is to work together. Life is good.

I’ll give you a hint

To maintain a high-functioning team, a key challenge is to avoid making comparisons between team members. Don’t give them any excuses to compete against each other. But how do you avoid comparisons? Use these simple pointers:

- **Describe what you like** An employee asks your opinion of some work: “How does this compare to Pat’s?” This is a trap, so don’t mention Pat. Instead, describe what you like: “That’s a great implementation. It’s clear and concise, easy to test, and uses all the right security protections.”
- **Express confidence in your employees** An employee complains about a teammate: “I’m blocked until Joe checks in his code, but he doesn’t want to check it in till it’s perfect. Can you get him moving?” This employee, perhaps unwittingly, wants to look good while making a teammate look bad, so don’t fall for it. Instead, express your confidence in your team: “Wow, you are blocked, but Joe’s trying to do a quality job. Sounds like a tough tradeoff. I’m sure you two can talk it through and work out a reasonable compromise.”
- **Focus on fulfilling needs, not on being “fair”** You promoted an employee and a peer is jealous: “How did Jane get promoted and I didn’t? It isn’t fair, we both worked hard.” It’s easy to get caught up in fairness on many issues, but life isn’t fair. Instead of comparing the haves and have-nots, reaffirm your employee’s needs and talk about fulfilling them: “I wish everyone could get promotions at once, but the business doesn’t work that way. Let’s talk about what you need to do for a promotion and put a plan in place to get you there.”

- **Talk about behavior instead of people** The employee you looked over for a promotion isn't satisfied and wants details: "But why did Jane get promoted instead of me? How is she better than me?" This is another trap, and one that is well supported by our review system. Focus on behaviors rather than people: "Jane demonstrated leadership of the feature team in designing the solution that was used for a key scenario and then fully implemented by the team. That kind of leadership is something you are still developing. Your design skills are strong, but you haven't shown the leadership necessary to get consensus on your design and see it through implementation."

Focusing on behavior also works well in review calibration meetings. Instead of saying, "Jane is better than Pat," you say, "Jane has reached this skill level as demonstrated in these ways. Pat has not shown those skills." While there is a comparison, it is not competitive. Pat isn't trying to be better than Jane; Pat is trying to attain Jane's skill set. It's a subtle but important difference.

One for all

Having a cohesive team pays huge dividends. They perform better, are more resilient, and have higher morale. Retention is better, communication is stronger, and the team is easier to manage. Being a good manager means caring about your people and other aspects I wrote about in my article, "I can manage" (which appears earlier in the chapter)—but those don't guarantee a cohesive team.

You need to keep your team from competing against one another. That comes from seeing your employees as individuals, understanding their potential, and focusing on their specific needs and your specific expectations for them. Care about your employees without judging them against each other.

When you remove individual competition, you only leave team success. Sure, there will still be quarrels and complaints, but your team will see those as inhibiting their shared success, not improving their personal standing. And nothing beats a team working together to make each other great.

By the way, we've had kids for ten years and still no signs of rivalry.

Eric Aside If you like behavioral books, also try *Don't Shoot the Dog* (Ringpress Books, 2002) written by Karen Pryor, a former dolphin trainer. She covers everything from quieting your dog to taming your mother-in-law. If you want something even more comprehensive for business, *Human Competence: Engineering Worthy Performance* by Thomas F. Gilbert (Pfeiffer, 2007) is the seminal text for broad and narrow organizational change. I talk more about Gilbert's work later in this chapter in "Culture clash."

March 1, 2008: “Things have got to change: Change management”

It’s the political season in the United States, making “change” a happy word around here. Politicians fight over who better represents change. They proclaim themselves to be agents of change. Hysterical admirers jump up and down waving “Change” signs. Change. Change. Change. As if change is desirable. As if change makes people happy. Nothing could be further from the truth. Are these people idiots?

At best it is temporary insanity. The concept of change is seductive, and if you want a new leader, then change is what you seek. But if people really wanted change, it would already be happening. Stasis is the basis for nice happy faces. The fact is that people hate change, as my friend U. R. Rong pointed out in “Stunted growth: Change phobia.”

Eric Aside U. R. Rong, a.k.a. James Waletzky, took over for me for a couple of internal columns during my sabbatical last summer. He’s got his own amusing blog called “Progressive Development” (*blogs.msdn.com/b/progressive_development/*).

Yes, change is inevitable, and embracing change is the right thing to do, but change rattles people to their core. It’s stunning the degree to which people will hold onto old ideas and familiar practices that make their lives miserable rather than change to improve their lot. So, good luck trying to fix things at work, at home, or in your community. People won’t like it.

A change would do you good

But what if a change is what you need? After all, keeping things stable is nice, but it doesn’t move you forward. To have an impact on your life, your team, and the world, you need to shake things up a bit. How do you get past people’s cavernous craving for constancy?

Glad you asked. It’s not easy, but here’s a five-step program:

- Make the case
- Prime the pump
- Test the waters
- Jump on in
- Keep it real

While most people do make the case, after that they usually jump on in, skipping steps two and three of the program. That's foolish. Skipping steps greatly increases your risk of failure. And don't give me that macho garbage about taking chances. Change is time-consuming and costly. Rolling the dice because you're too impatient or stupid to know better is no sign of courage. If you're putting in the effort, do it right.

Make your opening statement

Before you can even propose a change, you've got to make your case to your peers and management. I talk about doing this in detail in my column "Controlling your boss for fun and profit" in Chapter 8. Read it and get sponsorship for your idea so that your boss and peers won't work against you; they'll work with you.

If you fail to get sponsorship the first time, try another sponsorship route or quit while you're ahead. If you move forward without sponsorship, you'll get crushed. Making believe that it will all be fine and that everyone else is delusional is, in fact, delusional. You're not one of those geniuses that no one believed in. Those geniuses kept trying till someone did believe. If you proceed without backing, you're just an ignorant sap whose sob stories will fall on deaf ears.

Eric Aside Every day it seems I hear someone lament about how no one would listen or how management foiled their attempt at change for "no good reason." Yet a little digging turns up that these people never got sponsorship for their goals, or they didn't make their measures transparent so that people could see the problem and the improvement.

One thing I forgot to mention in "Controlling your boss for fun and profit" is that you should ensure sponsorship agreement on the measures, metrics, and targets for success. You can't get credit for reaching your goals without agreement first on what to achieve. Without that transparency, you can never claim victory. No one will notice. You can learn more about the right metrics to use in my column "How do you measure yourself?" in Chapter 2.

You are ready then?

Now that you have a plan of action, it's time to move forward, right? Wrong! Listen, this is important. Change is a painful process, literally. Change is closely tied to two volatile emotions—anxiety and grief.

Change is tied to anxiety because it is filled with uncertainty. Think about the last time you changed jobs. Were you concerned that the new job would be all right? That you'd be successful on the new team? That you'd have fun and enjoy it? Did you worry about what you'd do if it didn't work out? People tend to be uptight and on-edge during change; even humorless. Jumping right in and assuming that everyone will love it is just asinine.

Change is tied to grief because after the change the old ways are gone. The new ways are unfamiliar and uncomfortable. People initially long for the old ways. They miss them. It's grief, and you should know the five stages of grief:

- Denial
- Anger
- Bargaining
- Depression
- Acceptance

Jumping right in before people pass the anger stage is ill-advised. The bargaining and depression stages aren't much better. How do you avoid this debacle? By priming the pump.

Eric Aside "Priming the pump" is an expression that comes from pouring liquid into a pump to clear out the air and allow it to function better. It general, it means to take action in advance that helps you achieve a desired result.

I'm in my prime

Priming is exposing people to a situation before it actually occurs. Priming helps people perform better in new situations and feel more comfortable. If you've ever been to a rehearsal or practiced a speech, you've been primed.

Priming change means telling everyone what to expect in advance. Talking about what it will be like, in detail, before it happens. How long before? Long enough to get at least past the bargaining stage. Depending on the scope of the change, this could be a few days to a couple of months.

During the priming you'll want to talk about the change several times, perhaps a couple of times in e-mail and a few times in person. You'll talk about why the change is important in terms that matter to them. You'll talk about the inevitable difficulty of the transition, but lay out the clear path to the desired result.

By introducing the change in advance, you get people moving through their grief and anxiety before they actually are doing anything different. You also get to hear their objections (anger) and suggested alterations (bargaining) before the change goes into effect. That gives you time to make adjustments, and gives people a sense of control and ownership, all of which smooth out the transition. Brilliant!

They could use a good pilot like you

Of course, you'll never figure out all the trouble spots and hiccups with the change by just talking about it. You've got to try it out. How do you try it out and fix issues before everyone gives up on the idea? By testing the waters with pilots.

Pilots are trial runs of the change. You get together a small group or two to work the new way. Their special mission is to scout the future on behalf of the team. Ideally, these small groups are made of early adopters—flexible people who enjoy trying new things. You also want a couple of open-minded skeptics on the pilot teams—they'll find key issues and can become your biggest supporters.

Kick off the pilots shortly after you announce the change, while you're still priming the rest of the team. As the pilots discover issues and blaze the trail, the rest of the team has time to get through denial, anger, and bargaining. Remember, like all good scouting expeditions, pilots should document their discoveries so others can benefit. The pilots have three other advantages:

- **Experience** When the pilots have completed their initial work, your group will have a ready-made collection of experts on the change. Now you don't need to field all the questions and it won't seem like only your idea.
- **Ownership** In fact, as people on the pilot teams work past problems and have success, they feel more and more ownership of the change. It becomes their idea as much as yours, which is just what you want. Remember to be generous with your support and praise; it will serve you, your ideas, and your team the best.
- **Validation** By the time the larger team hits the depression stage, the pilots will return with stories of success. These quick wins are essential to get your group over their depression and into accepting the change.

Ready to make the leap?

When the bulk of the team has stopped raising objections and suggesting alternatives, and when the pilots have come back with quick wins, you're ready to jump on in. Because of your preparation, this shouldn't be a big deal. Most of the objections and problems will already be resolved. You'll also have the pilot teams there as guides.

However, don't expect everything to go perfectly. There will be more questions and issues that arise because of the increased scale, particularly in the first few weeks. New configurations and situations will crop up as more people take on the change. Let your team know to expect these problems, and that you and others are available to help. Again, broad and regular communication, both in person and through e-mail, is critical.

Eric Aside Of course, your change may fail for any number of reasons. That’s okay. I’d much rather see someone try and fail than risk nothing and learn nothing. This also points out another great thing about pilots—they allow you to fail early.

Gone but not forgotten

Once you’ve rolled out the change and people are starting to see the benefits, you might be tempted to celebrate. That would be premature and shortsighted. When you first got sponsorship, it hinged on achieving certain goals. Your ability to claim victory depends on reaching those goals. In fact, the survival of your change depends upon it.

How can this be? After all, you made the case, primed the pump, tested the waters, and successfully jumped on in. People seem to be getting used to working the new way and may even be enjoying their new-found success. Why would anyone want to switch back? Ah, but you forgot that the sponsors giveth and they taketh away. You must keep it real.

By “keep it real” I mean keep the value of making the change real to your sponsors. The moment they forget the impact of your change is the moment they take it for granted and allow someone new to reverse it. I’ll bet you know of examples at Microsoft and elsewhere.

This is why defining and tracking measures, metrics, and targets is so very important. The team and your sponsors can see the improvement over time. It never gets lost. Of course, when you eventually hit or even exceed your targets, you, your team, and your sponsors can celebrate the victory together.

There is nothing permanent except change

Change is a part of life. The people who control and manage it are the people who succeed. It’s not easy, but then again, it’s not that hard either. Anyone can do it. You can do it. Doing it well will grow you and your team. Prepare well, communicate well, and make your progress transparent. Before you know it, a change really will do you good.

June 1, 2009: "I hardly recognize you"

The annual engineering awards are being given out this week at the Microsoft Engineering Forum. Annual reviews will soon follow. These are great opportunities to recognize impactful work. It's too bad most managers are tragically ignorant of how to recognize their employees or truly why they should.

If you are a manager, you're probably relishing this opportunity to heckle all those bad managers. Guess what? I'm talking about you. You don't know how to use recognition properly. You don't know why you should. "But I'm great at recognizing my employees," cries a clueless manager. "I'm always congratulating my team—I even shaved my head once." Let's call this manager "Chaos."

Chaos thinks recognition is all about morale and motivation. Certainly, there are real benefits for Chaos being a team cheerleader. But if that's the depth of his use of recognition, then chaos is what he'll get.

Everybody wants results

What Chaos fails to recognize is that recognition is a form of reinforcement. Reinforcement drives behavior. Behavior drives results. Results are king at Microsoft. Good recognition focuses on reinforcing desired results (and correcting undesirable ones).

If you aren't thoughtful about the results you seek, recognition can easily drive detrimental behavior. For example, Chaos shaved his head when his team met a tough milestone. To meet that milestone, team members cut corners on quality and deferred fixing structural issues. Those issues increased the "bug debt," prolonged stabilization, and reduced release quality. More importantly, Chaos's team members learned that Chaos rewards cutting corners and doesn't respect quality.

Eric Aside Cutting corners for rapid development is desirable for prototypes and new ideas, where learning about the problem space is the result you seek.

The end may justify the means

"Yeah, but we hit the milestone. We built team unity. That means something!" claims Chaos. Do the ends justify the means? No, not with your narrow view of the ends.

The ends for Chaos were a united team hitting a tough date. However, other ends were achieved—increased bug debt. If the ends had been defined as a united team hitting a tough date with zero bug debt, then the means can be left to the team.

Chaos is getting cynical. He says, "So if the team robbed a bank to get money to pay a vendor to reduce bug debt then that would be okay?" No, it wouldn't. Chaos has introduced another end—jail time. Define the ends as a united team legally and responsibly hitting a tough date with zero bug debt, and that problem is avoided.

It's not easy to carefully define and communicate the ends you seek. It's much simpler to follow Chaos and reward haphazard results, ignoring the unintended consequences. However, setting clear expectations and recognizing your truly desired results pays huge dividends:

- You micromanage less.
- Your team innovates more.
- You get the results you desire.
- You have more time to focus on developing and recognizing the great work of your team.

The time has come to act and act quickly

Do you wait till the ends are achieved before recognizing them? Recognition is most effective when given immediately after achieving a goal. But you also want to recognize all the intermediate results that led to the end result.

For example, on the way to your united team legally and responsibly hitting a tough date with zero bug debt, your lead ran a great design review. You should immediately recognize her effort by saying, "That was a terrific design review today. I love how you listened to everyone's opinion without critique. I also liked how you summarized that feedback and improved the design. Correcting those errors and misunderstandings early will save us significant time and lead us toward hitting our date with zero bug debt."

Notice the keys to great recognition:

- **It's immediate** The same day, or even better, the same minute.
- **It describes precisely what you liked** As opposed to the generic "nice job."
- **It's tied to the desired end result** Reinforcing the real value of the effort.

You may not have the time or opportunity to recognize every positive step toward your end goal. However, you should keep your eyes open and address as many small victories as you can. Your team will love it, they'll better appreciate your expectations, and they'll be better aligned toward your desired result.

Let us celebrate

Chaos wonders, “Day-to-day recognition is great, but doesn’t it detract from the big celebration at the end? Was I wrong to shave my head?” Chaos’s grooming habits aside, a celebration marking a major achievement is just as important as day-to-day recognition. However, because it’s a culmination of a long-term effort, it’s not as simple as sharing a few comments or shaving your head.

First and foremost, recognition should be for a carefully defined desired result, not something arbitrary like “great effort” or “someone said something nice.” For example, don’t give out an award for “teamwork.” Give out an award for “teamwork across divisions and geographies that led to delivering a complete customer scenario.”

Once you have clearly defined criteria for the recognition, plan the celebration. Here’s what makes for a lasting positive experience:

- **Make it personal** If you defined the goal, the recognition should come from you. Include some personal remarks that express the spirit behind your goals and the way individuals or the team embodied that spirit.
- **Make it public** Public ritual is important. It carries meaning, builds community, spreads the message, and creates a shared sense of purpose.
- **Make it lasting** The associated award should be substantial to the senses—heavy, bold, eye-catching, and tactile. Shaving your head meets many of these criteria, but so do heavy physical awards (like the Oscars) and giant signed banners on heavy poster board.

These guidelines give your celebration enduring emotional impact. Food or money don’t cut it.

I’d like to thank the Academy

The last element to consider is “Do you give the recognition to the team or to the individual?” While you should certainly give the whole team a celebration and perhaps a thoughtful keepsake (like a poster for the team space), recognition for carefully defined desired results should be called out to a small number of recipients; typically, no more than five. Any more than that dilutes the impact.

Why is the impact diluted when given to more than five recipients? Because when you only recognize the larger group, there are sure to be individuals who didn’t embody the spirit of your goals. The message you send is that tagging along is just as good as being the driver. It isn’t. Who originated the action? Who drove it to fruition? Those are the people at the center and in the lead. Those are the people you single out and encourage others to emulate.

Eric Aside I was involved in the internal Engineering Excellence (EE) Award program for some time. It recognizes broad improvement in the way we engineer our products and services. The criteria are around demonstrating that improvement and making it available for others (business and customer impact, plus adoption). The recipients are those individuals who came up with the initial idea, first put it in practice, and drove adoption. The ceremony has ritual (Bill Gates used to host it) and the awards themselves are bold, heavy, eye-catching, tactile, and will last a lifetime.

Over the past several years, the EE Awards have recognized and encouraged dramatic improvements in our engineering—more secure and reliable products, better customer feedback, and broader language support. Reviewers in specialized areas are noticing the difference. Hopefully, when enough improvements are broadly in place, everyone will notice.

An interesting side note: a few years ago we gave an EE Award to a tool that consolidated a large number of duplicate efforts, gained broad adoption, and automated processes that improved our software. Unfortunately, the tool itself wasn’t well engineered, which hurt productivity and reflected poorly on the award program. A great example of needing to carefully define the ends you seek.

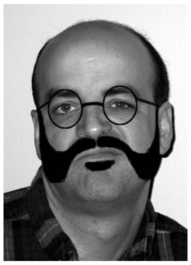
All right, let’s review

Now that you know how to recognize your employees and why you should, how can you apply this to the annual review process?

- Describe what you liked (and disliked) about the results your employees achieved and how they achieved them.
- Talk about the small results and the big results. Tie them together.
- Set carefully defined goals for the future that will drive personal development and stronger results for your business and customers.

Clarity around expectations tied to consistent feedback described in plain language is critical to getting great results, and a happy and secure team. Recognition requires careful thinking and deliberate action, but it’s not difficult to do. Avoid the chaos and you can even keep your hair.

October 1, 2009: “Hire’s remorse”



Looking for that perfect candidate to fill a role? Good, that means you’ll never steal a great candidate away from me. I love it when industrial-strength stupidity renders my competition comatose. You can’t hire the perfect candidate, but please keep trying. Maybe after six months I’ll even get your open headcount.

This isn’t a case of letting the perfect be an enemy of the good. Idiotic hiring managers aren’t trying to create perfect candidates—they are waiting for them to appear. It’s like a romantic awaiting their soul mate or a home buyer searching for their “dream house.” Guess what? Dreams aren’t reality.

If someone tells you on your first or second date that you are the man or woman of their dreams—run. They aren’t dating you. They are dating their dreams. Sooner or later, they’ll realize the two don’t match. The same thing happens when hiring managers get caught up in seeking their dream candidate.

Hiring people instead of pipe dreams

It’s natural for a hiring manager and their team to imagine what the new hire will be like. When you spend so much time checking resumes, doing phone screens, informationals, and interviews, you can’t help but think about the potential outcome.

Unfortunately, there are three dangers with getting caught up in fantasy:

- You may not interview a great candidate even if they meet your needs because they don’t match your preconceived notions.
- You may hold off on an offer to a great candidate because they lack attributes that were imagined but not essential.
- You may regret hiring candidates who turn out different from your projection of perfection, compromising their chances for success.

Of course, you shouldn’t hire people who don’t meet the high bar we set, but it’s awful to lose out on strong hires because you were busy hallucinating. Let’s dive deeper into each case.

Eric Aside I talk about informationals and interview loops more in “Out of the interview loop” earlier in this chapter.

I found myself much more reasonable

If you have a preconceived notion of whom you're hiring (worse case—someone just like you), then you're certain to subconsciously filter applicants based on that template. You may get candidates that match your musings, but you'll miss a crowd of candidates that meet your needs. If a good hire is one in a hundred (not far from the truth), the sooner you review a hundred qualified candidates, the sooner you'll hire someone. In addition, less filtering brings more diversity, which leads to better balance, better products and services, and better customer experiences.

If you hold off on an offer to a great candidate thinking that the next candidate might be even more ideal, you're likely to lose both. Surely another team will want your great candidate, and your next candidate will be at least as flawed. They say, "A bird in the hand is worth two in the bush." Thus, the next candidate better be twice as good if you hold off. Otherwise, make an offer now before your great candidate becomes your great regret.

Speaking of regrets, ever order something only to regret it shortly after? That feeling of buyer's remorse can afflict managers as well. Hire's remorse happens when you've imagined who might fill an important position, fill it with a real person, and then regret the decision because that person isn't precisely who you wanted. Quit sulking and get over it. Only movie directors occasionally get what they imagine, and that only lasts while the camera is rolling. You've hired a great person, now give them a chance.

Well, what do you need?

How do you differentiate what is really required for your open position from your unintentional biases? Consider the must-have results you seek from the role, and consider how soon you need those results. Then insist on candidates that have achieved those results in some general way. The sooner you need results, the more specifically the candidate needs to have achieved them before.

For example, say you need someone who can develop and ship code for a service. If you can afford to have them figure it out over nine months, hire someone who has demonstrated they can write code of any kind and ship it. It could be a game or a numerical algorithm for a thesis. All you need is the development skill and the ability to finish something. That could be almost anyone from any field—lots of possibilities.

However, say you're in SQL Server and need an engineering lead to step in right away. In that case, you need someone who has recently been an engineering lead. Your need for SQL Server expertise might not be as immediate though, so that's not an essential skill and shouldn't bias your thinking. You'd hate to pick a lousy lead with SQL experience over a great lead who can gain SQL experience over time.

The key is to focus on must-have results and the core skills essential to achieving those results. Then look for anyone who demonstrates those essential skills and gets analogous results. You can be more specific if your need is more immediate, but don't kid yourself. Getting picky may cost you candidates who'll deliver great results for years if you support them and give them a chance.

You could even say that he has principles

One item you generally won't find on a resume is a candidate's principles. Principles don't tend to align with appearance, skill set, or even achievements. Thus, when you imagine your ideal candidate you often don't think of principles, which is yet another reason not to waste your time imagining your ideal candidate.

What principles are important to you and your team? Integrity? Transparency? Accountability? Selflessness? Loose coupling? Screen for them. They are just as important as skills sets and demonstrated results. Many would argue that principles are more important. People can learn skills and work together to achieve results. Principles can be harder to attain or change.

Don't know what principles matter to you and your team? Figure it out—now. Let your team know. Being principled is itself a principle. For me, it matters the most and is the first trait I seek when talking to candidates.

I'm trying to tell you something about my life

Life is not a novel you get to write where everything turns out the way you conceive it. It's full of twists and turns for you to ride and hopefully enjoy in the end. Hiring and building a team is no different.

You have no idea who will respond to your open positions at what times, so don't try to guess. Instead, keep your mind open to all possibilities. Remember, you want a team full of differing viewpoints that are all aligned toward a common goal.

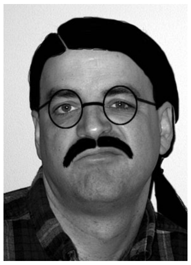
While you don't want unpredictable ship dates and quality, you do want unpredictable ideas and interplay on your team. They are the source of innovation and improvement. The more your team is filled with variety instead of clones, the more your team will enjoy learning from each other—a result reflected in more thoughtful designs and improved customer experiences.

So, consider the needs of your team, but stop short of projecting who might fill them. Keep your mind open to the possibilities, give offers immediately to great candidates unless you know the next candidate is twice as good, stick by your principles, and avoid hire's remorse.

Everyone complains about insufficient resources—the sooner you fill your open positions the happier and more productive your team will be.

Eric Aside This column was quite popular. It received special attention from recruiters and the corporate diversity group, who loved the message. Recruiting is time-consuming and can be frustrating, so I can’t say I adore doing it. However, I do enjoy being surprised by the wide diversity of talent we have at our company and in our field.

November 1, 2009: “Spontaneous combustion of rancid management”



What’s good for you isn’t always good for your group. Obvious, right? You can call it local versus global optimization. You can get geek philosophical about it and say, “The needs of the many outweigh the needs of the few…or the one.” Or you can simply notice the difference you feel between zany ideas from the intern (cool) versus zany ideas from your general manager (scary).

For example, spontaneity in an individual is a good thing and unvarying predictability makes Jack a dull boy. But when Jack is running a large enterprise, unpredictability can wreak havoc. There are managers who grow up and learn this lesson, and there are managers who are Randomizing Ambiguous Nimrods Causing Incessant Distraction, or *rancid* for short.

I despise rancid managers. They think they are responsive and flexible when in reality they are the fastest means to team dysfunction and failure. If you are a manager and tell your team, “Hey, I’ve got a great idea,” and they look back at you, seemingly saying, “Does it involve tying yourself to a tractor and driving off a bridge?” then you might be rancid.

It’s not that bad

What’s so bad about being a rancid manager? Isn’t consistency and predictability boring? The answer lies in Brownian motion, which describes the movement of particles under random bombardment. Particle movement in Brownian motion could best be described as erratic.

Consider a large team of engineers who are constantly being pushed in different directions at random intervals by their general manager. You’d expect the same erratic movement from that team. Their chances of actually accomplishing anything as an organization are negligible.

Alternatively, if the team is consistently pushed in the same direction they will gather momentum and make significant progress toward their goal. Therefore, a successful manager sets a clear direction, points the team in that direction, and consistently pushes in that same direction until the team's goal is achieved. Course corrections that naturally come along should translate to gradual team nudges, until the correction is made. Only severe circumstances should prompt major changes in course.

I'm talking about high-level direction, not the day-to-day details, which require more agility and flexibility. The key is aligning the day-to-day detailed decisions to the high-level direction. You can't do that if your high-level direction constantly changes.

Eric Aside "But shouldn't I push my people to work outside their comfort zone?" Yes you should, but that is about risk management. It doesn't mean randomize your people. The idea is to always carry enough risk to scare people, but not so much that people freeze or panic. This creates challenge and tension, which drives high performance. As I talk about in Chapter 1 (particularly "Right on schedule"), you constantly need to adjust your risk as situations change. If a scary dependency comes through and the risk drops, you might consider adding back some bold features. If a key team member must go on leave, which raises risk, you might consider dropping a dependency or a significant portion of functionality. Regardless, carrying a healthy amount of risk is good. Being a rancid ninny isn't.

Do I look all rancid and clotted?

If teams run by rancid managers are erratic and make little progress, why are there so many rancid managers? Here are three reasons:

- **Being rancid seems responsive** Instead of always pushing in the same direction, rancid managers respond quickly to outside influences, giving their management a nice warm phony feeling of agility.
- **Being rancid means never having to achieve anything** Since the world of a rancid manager's team is always changing, there's a self-fulfilling excuse for why nothing gets accomplished. ("Hey, a bunch of stuff happened!") If these managers weren't rancid, they'd risk being accountable. They'd have to do the hard work of setting an achievable vision and direction for their team and then achieve it.
- **Being rancid keeps you occupied** Where's the fun in consistency? What do you do all day to stay busy? What value do you bring to the team if you're always saying the same things? Making a call and sticking to it is difficult. It's scary, and you might need to do real work to support it.

Of course, rancid upper management breeds rancid middle management. It's hard to escape. Often people don't even know how good consistent management can be until they finally experience it.

It's a path made of principle

When managers stop being rancid and start being consistent, a number of wonderful things happen. First and foremost, teams start achieving results. Managers move from creating havoc to preventing havoc. And team members actually understand what they are supposed to do each day, and know they have to do it because it won't be different tomorrow.

All this leads to a higher sense of purpose and higher morale. At first, the team may struggle to get fully aligned, but soon the job of a manager becomes easier. You're not always changing your story. Once the team is aligned, there are fewer problems to resolve. It's easier to track and show progress.

What challenges does this leave for managers? The biggest is to paint the picture of where the team is going. You need to be clear and understand it deeply in order to describe it consistently and repeatedly to the team. Once you have your story of the future, the challenge is to stay on course at a high level as team members and external factors change. You must discover and communicate adjustments as needed, and prevent upper management from inappropriately disrupting your course.

It's not easy. You need confidence in your convictions and strong thinking behind your direction. It helps to have principles you follow and communicate. They let people know what you expect and how you make decisions. Doing so helps them align their decisions with yours and gets the whole team behind you.

Eric Aside Upper management edicts can sometimes be hard to discern. Are they appropriate customer- or business-driven changes that are worth disrupting your team's course, or are they inappropriate distractions that you should filter from randomizing your team? The best way to find out is to ask your manager or a trusted mentor within your organization. Be direct. Find out the background and thinking behind the change. Even if the change is distasteful, it might be the right thing to do. Even if the change is alluring, it might be wrong to follow.

Picking your battles is a critical skill to master if you hope for a long and prosperous career. Understand the context. Learn who you'd have to fight, how tough it would be, and how much you'd stand to gain or lose. Then make an informed decision about whether the battle is worth engaging. Discretion can add years to your valor.

Oh, the noise! Noise! Noise! Noise!

"But things change!" protest the rancid managers. "Is your team supposed to blindly follow some ancient plan? Our agile competitors will beat us every time." Of course things change, and plans, architectures, and designs must adjust. Details and understanding constantly evolve, driving continuous iteration of our work. But if your goal today is to build a social computing experience, no detail or external influence should be switching you to build tractor tires.

Strong managers know that all kinds of variation in team structure, competitive landscape, market fluctuations, and technical challenges precipitate changes to the best laid plans and designs. However, it's rare for those variations to necessitate a complete shift of direction. A strong manager welcomes change and contextualizes it for their team so that their momentum toward a shared goal stays strong through many corrections in course.

Eric Aside Great managers directly confront distractions to keep them from disrupting their teams. Some have a regular "rumor mill" portion of their staff meetings to discuss the latest gossip. Shining a light on these issues and nipping them early keeps teams focused.

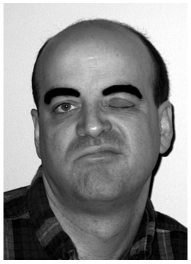
All progress has resulted from people who took unpopular positions

Being consistent as a manager may seem dull, but it's the kind of dull that teams truly appreciate—the kind of dull that leads to results. While it sounds easy to be consistent, it requires courage. Courage to stand your ground and stand behind your words. Courage to be accountable for the direction you've set.

I'm talking about commitment. I'm talking about integrity. I'm talking about shaping the world to fit your vision rather than mindlessly following the latest trend or circumstance.

Our vision should be shaped by the needs and aspirations of our customers and the imagination of our collective minds, but in the end our vision belongs to our leaders as individuals. Our success depends on them having the courage to speak it and steadfastly pursue it. Do you have that courage?

January 1, 2010: "One to one and many to many"



Does the prospect of a one-on-one with your manger make you energized or anxious? Are your morale events packed with peers or attended only by slackers and scandal spreaders? Chances are that one-on-ones are at best bearable for you and morale events are rare, wasteful, or both.

Wasting one-on-one time and morale events is inexcusable. It takes what could be the most valuable time of your week or month and turns it into redundant, useless, and pathetic time-sucking, cash-burning, work-interrupting guilt trips. ("Shouldn't I be enjoying this?!?") Is there anything you can do about it? Yes, there is. Have your manager read this column, because it's all your manager's fault.

Am I speaking to your manager now? Good. Hi, manager. You stink. You're not using one-on-ones and morale events properly. It's all your fault. Only you can fix it. However, all is not lost. Fixing one-on-ones and morale events is simple, inexpensive, and can turn a collective time furnace into one of your greatest tools for team improvement.

A deeper purpose

Let's start with understanding the true purpose of one-on-ones and morale events. One-on-ones are not for status updates. You get status through charting and status meetings. Morale events are not for manipulating people into believing they are valued. People believe they are valued when you actually value them through trust, guidance, and personal acknowledgement.

Eric Aside You can talk about how projects are going during one-on-ones, but only if that's what your employee wants to discuss or you have some time left over. If you really need to have a detailed status update from an individual, for whatever reason, drop by or put yourself on the employee's calendar. After all, you are the boss.

- **What is the true purpose of one-on-ones?** To develop strong trust relationships with your staff and get to know them on a personal level. Management is all about effective delegation, which is all about trust. Trust comes from mutual respect and understanding, which comes from knowing each other as human beings and nurturing a relationship. You also need integrity, which is necessary for nearly everything of value.
- **What is the true purpose of morale events?** To break down barriers between people and teams by humanizing them to each other. People working together inevitably creates conflict. A business environment tends to dehumanize people, which fuels conflict. You hear phrases like "They don't care" and "He's an idiot." By taking people out of the business environment to interact in fresh ways, you put human faces to former pronouns and create relationships that lead to improved collaboration and understanding.

Now you know the purpose of one-on-ones and morale events. Next, let's talk about how to run them effectively, starting with one-on-ones.

Between you and me

One-on-ones can be anywhere from 30 minutes to 2 hours. I typically set aside an hour weekly for each of my directs and 30 minutes monthly for every one of my skip-level folks. Before you faint, do the math. Monthly skip-levels with everyone take about half as much time as weekly one-on-ones with directs. There's no excuse, you should do both.

Why run one-on-ones so frequently? Because you miss them. Several times a month you get unavoidable conflicts. If you miss a weekly one-on-one, you'll still talk to the person every two weeks. However, if your one-on-one is biweekly, you end up talking once a month, which is too infrequent to maintain a trust relationship with your directs. The same argument applies to skip-level meetings.

What do you talk about? Whatever your employee wants. **One-on-one time belongs to the employee.** When is your time? Whenever you darn well please. You are the boss. You can march into anyone's office at any time to get whatever you need. One-on-one time is for employees to get what they need.

Eric Aside Okay, wasted one-on-ones are not all the fault of managers. Employees do have to consider what they need and engage their managers. However, managers enable that discussion by resisting the urge to dominate the meeting.

"But what if the employee wants to talk about alpine wildflowers? Isn't that a waste of time?" No, it isn't. Remember, the purpose of one-on-ones is to develop strong trust relationships with your staff and get to know them on a personal level. What better way to get to know someone and develop trust than to talk about wildflowers? Look around your employee's office. It's loaded with potential topics displayed on the walls or the desk. Worst case, you can always talk about your weekend.

Eric Aside While some employees are happy to talk about their weekends and personal interests, some are more protective. You must respect their privacy and never pry. That's why it's usually safe to ask about the pictures on their desks and walls. They are topics openly on display. If you want to talk about weekend activities, always briefly start with your own and wait to see if your employee is willing to reciprocate.

Float like a butterfly

Aside from building a relationship, why is it so important to talk about personal interests? Allow me to relate an example.

I once had a lead reporting to me who loved remote control (RC) helicopters. Each year, he and his fellow enthusiasts would gather at Nationals to compete and set records. We'd talk about it frequently at one-on-ones.

One year we were shipping around the weekend of Nationals, and some problems arose. The lead offered not to go, but everyone was well prepared to cover for him. He went, had a great time, and we got the product shipped. We knew about the conflict and were prepared well in advance because I knew how much Nationals meant to him. As you can imagine, the lead was quite grateful.

Consider what would have happened if the lead hadn't told me all about Nationals months before. Who would have agreed to slip because of a RC helicopter convention? The lead would likely have canceled his trip. He'd understand and keep his spirit up, but deep down he'd be disappointed and probably resentful. Compared to the tremendous appreciation the lead actually felt, it's quite a shift in attitude.

Now consider that this example was about competing at a national convention, a desirable event. Imagine the impact of negative events, like family illnesses, divorce, or other tragedies. The employees entrusted to us as managers are whole human beings with lives at and away from work that impact their performance on the job. Ignoring their life interests and issues puts you and your team in jeopardy. Embracing them builds loyalty and trust.

How am I doing?

"But when do you discuss career development or performance issues?" One-on-ones are a great time for coaching and career development. Usually, you don't have to bring this up with employees. They often want to spend time talking about advancing their skills and position. However, you should always be looking for opportunities to highlight critical decisions, strategic and systems thinking, change management, communication style, and other situations and work products that might be used to improve an employee's capability.

In the unusual event that you don't talk about career development and commitments with your directs over the course of a month, be sure to bring it up yourself. It's too important to leave entirely to chance, and no one likes to be surprised by feedback that should have been delivered earlier. This also means you should be thoughtful and prepared to give feedback whenever an employee asks.

In particular, performance issues can't wait for "teachable moments." They need to be handled immediately and the messages reinforced at one-on-ones. Depending on the seriousness of the issue, you may need to document your feedback in writing so there can be no mistaking what you mean and when you meant it. For further thoughts on this, read "The toughest job—poor performers" earlier in this chapter.

Are we having fun yet?

Enough about one-on-ones—what about morale events? Remember, the purpose of morale events is to break down barriers between people and teams by humanizing them to each other. How often is this necessary? Well, how long does it take to objectify people? I think you need an event every month.

Monthly morale events keep people treating each other like people. Of course, you can schedule one month as a team event, followed by an event for the larger group, followed by a quarterly event for the entire organization. As long as people are engaging in a humanizing way at least once a month you should be covered.

"But isn't that expensive?" No, because the best morale events are often the cheapest. Ideally, the activity is an equalizer—something no one can dominate or where experts aren't taken seriously, like bowling. Here's a short list of cheap and easy diversions:

- Frisbee golf (free)
- Board games—Trivial Pursuit, Apples to Apples, and Scene It are ideal team games
- Card games, particularly games that involve four or more people
- DVD watching with take-out food
- Geocaching (*geocaching.org*)
- Billiards and ping pong at a local pub
- Hikes or other outdoor walks

Remember, when it comes to morale events, cheap beats infrequent every time. Big teams can afford to spend more on activities like bowling, whirlyball, or curling. The key is to break out of your collective rut and have fun together like real people.

Eric Aside Morale events are best avoided during crunch times even though they may be needed more than ever. The trouble is that people are stressed and won't easily relax. They'll be heavily distracted and not let go. Many team members won't participate at all.

The best solution to crunch times is to minimize their frequency and length through solid planning and prioritization.

You gotta trust me

If one-on-ones and morale events have anything in common, it's about taking time out to deal with each other as whole human beings. Doing so builds understanding, friendship, and trust. The relationships generated improve communication, collaboration, and team unity.

It's not that people become better talkers and writers when they know each other. It's that mistakes and misunderstandings are more easily forgiven. Context is better appreciated and acknowledged. People try to figure out what someone really meant and are willing to talk it through. When people work with trust and understanding, we operate as a greater whole and achieve more together than we possibly can apart.

July 1, 2010: “Culture clash”

Culture is management’s boogeyman—a monster that can’t be controlled, an immoveable object that can’t be overcome. If you ever want to see managers become whiny, petulant infants, ask them to challenge an issue ingrained in the corporate culture.

When managers say, “That’s a cultural issue,” they mean, “There’s nothing I can do about it.” When managers say, “You’d have to change the culture,” they mean, “I’m wimping out on this issue.” Culture is a manager’s favorite excuse to maintain the status quo and avoid confronting change. That is not only lame—it is naïve and indefensible.

Culture as concept seems interwoven into the very fabric of our lives. What hope would anyone have to change such a thing? Yet corporate culture doesn’t arise in a vacuum. It is instilled by the people who founded the company and changes gradually as the people change. Therefore, you can affect culture, and it can change. You simply need to know how to do it, and stop being a wimp.

You see the whole culture

Countless books, articles, blogs, and commentaries discuss what drives and sustains culture. It basically comes down to expectations, modeling, and rewards.

Your parents, peers, neighbors, and friends expect you to behave a certain way. They model it for you. They correct or shun you when you fail to comply. They applaud and encourage you when you set the “right” example. This is how culture perpetuates itself.

Can you change culture? Sure—simply expect, model, and reward different behavior. Really, it’s that easy.

Ever notice how the culture of a group or an organization changes when the leader changes? That’s what’s happening—the new leader expects, models, and rewards different behavior.

Eric Aside The behaviors a new leader expects, models, and rewards aren’t necessarily the ones the leader intends. The leader may say one thing, but actually expect, model, and reward another. The old saying, “Do as I say, not as I do” is pretty ineffective at driving cultural change.

Back off, man. I'm a scientist.

Is there a science to driving behavioral change in organizations? Of course there is—it's called Human Performance Technology (HPT). You'd think every manager would know about it, but few do. Maybe they prefer whining and impotence.

The basis of HPT is B.F. Skinner's model of operant conditioning. Basically, this model says that voluntary behavior breaks down into stimulus, response, and reinforcement. You receive a stimulus, like your manager telling you to start coding. You respond by coding. You receive reinforcement, like pizza.

Thomas Gilbert analyzed the types of stimulus, response, and reinforcement in his book on Human Competence, differentiating between the environment and the person.

- The environment refers to workplace influences, like expectations that stimulate a response (such as instructions and metrics), tools and processes that shape the response (such as build systems), and incentives that reinforce the response (such as bonuses and recognition).
- The person refers to individual influences, like knowledge to interpret a stimulus, capacity to respond, and motivation to receive the incentives. (If you're not hungry, pizza isn't an incentive. Hunger is the motivation.)

To drive behavioral change, you need to consider and potentially shift the environmental and individual elements of stimulus, response, and reinforcement, known as the "Six Boxes."

	Stimulus	Response	Reinforcement
Environment	Expectations	Tools & Process	Incentives
Individual	Knowledge	Capacity	Motivation

Eric Aside The concept of putting Gilbert's six areas into plain language as "Six Boxes" comes from Carl Binder (*http://www.binder-riha.com/sixboxes.html*).

Environment

Expectations: Suppose you want your software development team to focus more on unit testing, perhaps even adopt Test-Driven Development (TDD). From the environment side, you need to ensure that writing unit tests is expected by management and the team. It would be helpful to have metrics that indicate how much of the code has unit tests in place. However, that's not enough.

Tools & Process: The team will need to have a good unit testing framework in place—one that is easy to run, easy for developers to add tests to, and easy to use for measuring results. Of course, that's not enough.

Incentives: If group managers reward developers for checking in code quickly, with or without unit tests, developers will stop writing the unit tests. To drive unit testing, only developers who write thorough unit tests should be chosen for recognition and rewards. But that's not enough.

Eric Aside Which reward works best? Typically, it's not salary. Salary can actually demotivate when people aren't paid fairly. Also, recent studies show that salary doesn't align well with what people truly desire when doing difficult work—autonomy, mastery, and purpose (check out Dan Pink's talk on the subject at *www.youtube.com/watch?v=u6XAPnuFjJc*).

Instead of more money, offer more responsibility and opportunity to improve, and tie people's work to outcomes that matter. These rewards better target engineers' motivations and also naturally lead to raises and promotions.

Individual

Knowledge: Developers will need to know what the different kinds of unit tests are and how to write them well. They will need to know how to use the unit testing framework, how to get metrics on their progress, and what targets they are expected to reach. Now, we're almost there.

Capacity: Naturally, the developers will need to be capable of writing unit tests. Ideally, in this particular case that's not a problem. However, you might also expect your test team to unit test its automation. If your test team is using an automation tool, the team members may not be capable of scripting it or writing external code to test it. You might need to hire testers who have that capacity.

Motivation: Finally, your developers need to care about meeting these new expectations and receiving the recognition and rewards you offer. If your developers are completely apathetic or entrenched, they won't be motivated sufficiently to make the change.

Is it deliberate?

I've outlined a whole bunch of system thinking for a simple cultural change. You might say that's overkill. However, look back. If any one of the six boxes isn't properly addressed, the change won't happen properly, if at all. "I know—we'll just train everyone" isn't enough.

The environment impacts everyone and thus is more potent than any individual. No amount of training will overcome the wrong expectations, tools, and incentives. However, clueless, incapable, and apathetic engineers are also quite debilitating.

Review all six boxes. In the unit testing example, you'll see we need clear manager expectations, metrics, targets for those metrics, a unit testing framework, a change to rewards and recognition agreed upon by all the group managers, training for the team, and the right team members in place—both from a capacity and motivation perspective.

How do you think all that through in advance? Easy—know where you are today, know what you want to accomplish, and then go through the six boxes with that gap in mind.

You must avoid jumping to the first sexy solution ("I know—we'll use that new tool!"). You must be deliberate about your goals. You must also be deliberate about measuring progress toward achieving your goals, not just making pretty charts.

As I covered in "How do you measure yourself?" in Chapter 2, measuring lots of values and putting them on graphs achieves little. The numbers don't have meaning and the metrics will be gamed. You must know what you want to accomplish and deliberately measure your desired end results.

I'm going to ask you a few questions

HPT also provides a clear framework for performance improvement in general. Before proposing a solution, HPT requires that you understand the problem, the desired result, and how to measure success. The next time someone runs into your office suggesting that the team adopt a new estimation method, patiently hear him out and then ask him a few questions.

- What are you trying to accomplish?
- How is it done today?
- How will you know when you've accomplished your goal?
- What should managers say and do to support this direction?
- What processes and tools will the team need?
- How will managers need to recognize effort and progress differently?
- How will the team learn about the new direction, tools, metrics, and targets?
- Are the right people in the right roles to accomplish your goal?
- Does the team care at all about this?

All together now

Remember, if you are serious about doing things differently, then you are working on changing the culture. This applies to big changes, like moving to a services model, and little changes, like using code review checklists. Of course, big changes take longer than small ones, which is why goals, metrics, and targets are so important.

Changing the culture is possible and even algorithmic. When you ask the right questions, focus on the result, and consider all six elements that drive behavior, you can move molehills as well as mountains.

Chapter 10

Microsoft, You Gotta Love It

In this chapter:

There are three sides to management: the people side, the project side, and the business side. I've already covered people management and project management. It's time to tackle how Microsoft runs its business.

It's easy to be cynical about capitalism and huge corporations like Microsoft. Microsoft must turn a significant profit to stay in business. The company can't make decisions based purely on idealism or what's best for the customer. Staying in business must be a factor or we soon won't have customers. Unfortunately, idealists don't last long in a big company. Luckily, I'm pragmatic and willing to live with some imperfection if it means I get to spend more time with my family. Of course, I still get to ridicule the inevitable foibles of the world's largest software company.

In this chapter, I. M. Wright analyzes and overcomes the challenges of running a successful software business. The first column finds hidden merit in reorganizations. The second identifies the proper role of middle management. The third column accepts the challenge of directing one of the world's largest engineering teams. The fourth compares Microsoft's business strategy with that of Google and other competitors. The fifth examines Microsoft's growing pains and suggests ways to smooth its transition to middle age. The sixth reflects on the dysfunctional view of innovation in Microsoft's culture and how it should be corrected. The final column discusses Microsoft's changing approach to organizational structure and what it means to our business decisions and execution.

Given my monthly ranting, it's a common misconception that I don't like Microsoft. People think I like getting paid, but if I was in charge I'd change far more than I'd keep. At the risk of alienating all the Microsoft bashers out there, I must be honest. I love Microsoft. Having worked for academia (RPI), the government (JPL), small *business (GRAFTEK), mid-size business (SGI), and large business (Boeing), Microsoft is by far the best place I've been. For all its faults and gaffes, Microsoft has three amazing strengths:*

- *The people in charge earnestly want to make the world a better place…through software. At every other company I've known, the approach is the opposite: be successful in a field and hope it improves the world.*
- *The people in charge hire the best and then fundamentally trust them at the lowest levels to make decisions and run the business. Naturally, this has led to some problems over the years and forced checks to be put in place, but empowerment is more than a nice buzzword.*
- *The people in charge embrace and encourage change and then have the patience and perseverance to see it through. So many companies fight to keep their comfortable status quo. Microsoft fights its size and legacy to constantly improve and adapt to the world it helped shape.*

I love Microsoft and work every day to make it better. It's not perfect, nothing manmade ever is, but if I were in charge, I'd build from its strengths, not refashion it from my own imperfect image.

—Eric

November 1, 2001: "How I learned to stop worrying and love reorgs"

This issue of *Interface* talks about some of the technological changes and opportunities for delighting our customers that are rising from the .NET initiative. But the changes haven't ended there for most of us. I'm speaking of the dark side.

Yes, it's the dark side that comes seemingly with each and every change in company and group strategy: reorgs, frigging reorgs. Frigging wastes of time, energy, and momentum. Just the physics of them makes you want to duck, cover, and hold onto your desk for fear of it being moved to another building.

Eric Aside This column on reorganizations (reorgs) is targeted at managers, as are all the early *Interface* columns.

Down the Tower of Babel it goes

As the reorg message travels down successive layers of management, it gets diluted until finally the mail from your boss says, "Just keep doing the great work you're doing; nothing changes for now."

Of course, things will change for your group within three to six months. Maybe you'll simply move offices; maybe your management team will change; maybe your group will merge with another or just be cut loose. Who knows? Certainly not your staff, because you wouldn't want to tell them even if you did know. Why? Because plans could change five or six times between when you inform your staff and when change actually happens, and because of all the unease and distraction it would cause your folks.

Eric Aside You shouldn't hide it from them either, as I discussed in "To tell the truth," which appears in Chapter 1. If anyone asks about reorg rumors, you tell them the truth: "I have no idea what will happen or whether anything will happen at all. I do know what will happen if we can't stay focused on the work at hand."

Life in hell

Meanwhile, your life as a manager becomes a living hell. Even if you don't tell your staff about the coming changes, rumors start flying. Soon you can feel the panic building in your junior staff, while cynicism leaches into your senior staff. In addition to actually doing your normal 60+ hours per week job, you need to

- Calm and reassure your junior staff.
- Convince your senior staff to care about the changes.
- Grapple over offices and seniority on move maps.

  Eric Aside Two quick pieces of advice for office moves: avoid the southwest corner of a building, it gets hot; and use a purely objective, transparent method for choosing office assignments. (Microsoft uses time on the job, also known as seniority, down to the second you signed.)

- Play political death match with new groups that cross yours.

  Eric Aside If two groups merge, each can have only one product unit manager (PUM), group program manager (GPM), dev manager, and test manager. A highly politicized battle can ensue.

- Conspire with your manager to rephrase your group's current plans to match those specified by the new upper management.
- Fight the tendency to play philosophical Ping-Pong with your conscience and peers over making the right changes to your products that match the new company direction as opposed to meeting your ship dates.
- Provide yet another educational series for your new upper management about what your group does.

How can this much nonsense be worth it? Why must we go through organizational self-mutilation every 9 to 18 months? I've got a theory, and it's not the classic "We must always keep at least one large group moving because we don't have enough office space" theory. Here goes…

The road less traveled

What makes big companies like IBM and Boeing as agile as a cashier on quaaludes? IMHO, old orgs are like old habits taking the path of least resistance. It's hard to teach an old org new tricks. Why? Because people deal with folks they know much faster and easier than with those they don't. If an org has been around awhile, no matter how sharp the people are who run it, they will still tend to work with the same peers repeatedly, rather than deal with someone new.

This is a recipe for making middle management stagnant and ineffective. The longer the same managers are in the same org, the more likely it is that they will make decisions based on people they know instead of what's necessarily best for the company or the customer. It's an insidious disease that infects even the smartest people and goes completely undetected. Their lives seem easier and more familiar, while their choices become more restricted and perverse. Total disaster.

How do we keep our managers alert to new possibilities and away from following old habits? We move them around—constantly. Short of getting rid of all middle management, which may seem tempting but really isn't reasonable, the only way to keep our big company acting like a small one is to keep people moving—literally. Yeah, there's pain in dealing with misinformed or apparently clueless new management, but it beats the "old boy" network every time for agility, flexibility, and forward thinking.

Eric Aside The other solution is to flatten the organization—that is, remove layers of management by having larger numbers of people reporting to each manager. Microsoft has flattened a bit since I wrote this column, and a few groups are experimenting with flattening in the extreme. Going too far concerns me for the same reasons I don't have 15 kids—lack of attention and lack of oversight—but it's worth trying and learning from the result. You can learn more about Microsoft's latest approach in the last column in this chapter, "Are we functional?"

Of course, reexamining what you are doing and trying to explain how it fits into the corporate vision is a healthy exercise, and learning about people and projects in different orgs is always a good thing. Thus, I have learned to stop worrying and love reorgs.

Part of the problem or part of the solution?

Sure, we are in a highly competitive market where change is constant and new technologies, architectures, and platforms require us to adapt our strategies. But if all we do is shift focus and not people, our company will get organizational arthritis and stiffen till it becomes a corpse.

The question is, are you or your group part of the problem? If you've been in the same org too long, maybe things seem a little too comfortable and maybe it's time to switch. If your boss has been in the same org too long, maybe it's time to worry.

Regardless, the next time you get that mail from SteveB saying, "I'm as excited as I've ever been about the opportunities these changes will create," suck it up and say, "Steve, I'm as excited as you. Keep up the good work and bring on the new org!"

March 1, 2005: "Is your PUM a bum?"

At the risk of insulting some of my wonderful, former bosses, I think most product unit managers (PUMs) are bums. They pace the hallways, spewing crazed, detailed theories of how things should be—completely out of touch with reality—while living off the kindness of strangers who work in nearby offices. What value do they bring? What purpose do they serve?

Eric Aside I certainly don't want to insult bums with this comparison. Only a fraction of bums are mentally disturbed. Many live independently in the life they choose. As for PUMs, keep reading…

PUMs are the first level of managers truly removed from any real work. They don't write the specs or the code or the tests or the content. They don't localize or publish or operate or design. They just manage the people who do, along with their budgets. Managing multiple disciplines and their budgets is challenging, but so is dealing with a disconnected, delirious, and demanding boss. Neither should be a full-time job.

Eric Aside Kudos to a former team member, Bernie Thompson, who suggested this topic and now runs an interesting site on Lean software engineering with another former team member, Corey Ladas.

The man with a plan

Of course, your typical PUM will tell you he serves a critical role. For convenience, we'll name him "Clueless." Clueless spends hours in meetings discussing business development strategy. He fosters key strategic relationships with partners and customers. Clueless discusses these strategies at his staff meetings and at offsites. He presents his three-year strategy quarterly to his boss, VP, and overall team. "Strategic planning is critical to our success," claims Clueless. "Now all we need to do is execute."

And yet, Clueless seems somehow detached. Perhaps it's because he isn't involved with the day-to-day struggles of the team. Perhaps he doesn't receive enough information from his staff. Perhaps he's more passionate about being a PUM and a leader than he is about the actual work. Or perhaps he's so wrapped up in some abstract strategy that the means of shipping the product seems superfluous. Clueless had it right when he said, "All we need to do is execute." The trouble is that he left himself out of the equation.

I can't wait to operate

Every business plan has two sides:

- **The strategic side** What are we going to accomplish, for whom, and when?
- **The operational side** How are we going to accomplish it?

Clueless has the strategic side covered. For someone who isn't going to do the work, the strategic side seems more fun and relevant. Setting a strategic vision is important and must come from the top, so Clueless feels wanted and needed when he focuses on strategy. As for the operational side—well, he can leave that to his competent staff.

But theory without application is merely self-gratification. Architects who don't stick around to solve issues when the building is built or the software is coded are soon marginalized. And strategists who ignore the operational side become nothing more than figureheads who can't understand why the strategy isn't working out the way they planned. In other words, they become Clueless.

The devil is in the details

The strategy is set; yet six months into the project Clueless can't understand why things are already deteriorating. So Clueless starts attending all the meetings, asking questions, demanding results, and driving his staff and his team completely insane.

Micromanagement can get the job done if Clueless is sufficiently obnoxious and amphetamine-rich. However, he will be despised and with good cause:

- **No one likes to be treated like a baby.** Clueless has a good staff; he should let them do their jobs.
- **Everyone curses the bottleneck.** When Clueless gets involved in every aspect of the project, all decisions must go through him and he becomes a factor that blocks progress.
- **Opportunities for growth dissolve.** Clueless is taking on all the decisions and all the risk. He's shut out everyone else from stepping up, while he gets all the credit. Yay, Clueless!
- **The big problems are left unresolved.** By focusing on the details, Clueless sidesteps the big problems related to how the team operates and he leaves them broken. The entire team becomes dejected and demoralized. Ignore improving the engineering system and the entire project is set up for failure.

There's a better way for a PUM to drive a team to successfully execute on a strategy. It's called *an operating plan*.

The rules of the road

An operating plan describes how your team is going to execute on the strategic vision. As a PUM, you don't need to lay out all the details and make all the decisions, but you do need an operating plan that will adapt and adjust as the project iterates toward the desired goals. Just like the strategic plan, the operating plan should be built by the PUM in conjunction with her staff. Before the strategy is finalized, the team leadership needs to know how they're going to implement it.

For a product unit, the decisions are around people, processes, and tools. Here is a sampling of the kinds of questions you want answered:

- **What will your org look like?** Who are the leads? Who are the feature teams? Who are the experts and architects? What is the succession plan if people leave?

- **What processes are you going to follow?** What are you going to do better this time? What's your scheduling process? What are your quality goals, and how do you intend to hit them? How will decisions get made for triage and changes to the plan?
- **What tools will the team use?** What's your build system? What improvements are you making? What languages and components will you use? How will you automatically and objectively measure your quality goals and give the team feedback on its status?

When an operating plan is in place, team members can easily get the information they need to get on track and stay on track. The PUM can quickly understand the status in context and know where to focus her attention and direction.

Back on course

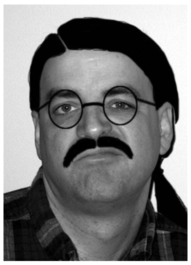
Both the strategic and operating plans will need adjustments over the course of a long project, but the team will always be in a position to succeed. That's because the strategic plan provides the vision and priorities that tell them where to go, and the operating plan provides the goals and limits that tell them how to get there.

People who do the real work, like you, often complain about not getting management support for making improvements or doing things right the first time. Well, were those improvements part of the operating plan? Is your PUM holding the team accountable for executing on that plan, or is the team only accountable for shipping all the PUM's favorite features on the date he told his boss? Don't just complain—point out the gap and take action.

Is your PUM a bum? Does he just provide strategic direction without also setting clear goals and limits? When things go wrong, does he become Clueless? If so, it's time to get your PUM to step up and define not just what he wants done but how he wants it done. Provide him with your guidance and wisdom. Help him make good decisions about what team goals to measure and what tools to use to drive your success. Then your PUM won't be a charity case, and you'll have all the support you need to do things right.

Eric Aside Six years later, PUMs are practically a thing of the past. Divisions that used to be collections of small, related product organizations are now single, large functional organizations. It started in Office and then spread to Windows, Developer division, Dynamics, Windows Phone, Bing & MSN, and most recently my own division, Interactive Entertainment. The only large division left with PUMs is Windows Server, which I'm guessing will change within a year. I discuss the shift to functional organizations and what it means in the last column in this chapter, "Are we functional?"

September 1, 2006: "It's good to be the King of Windows"



The Windows Vista project is nearing its end. It seems like a day doesn't go by without someone announcing they are moving to a new group and someone else announcing, "I am tremendously excited to be joining [fill-in your favorite organization]."

Yes, life is uncertain, and change leads to ambiguity and chaos, but I love reorgs. I even wrote a column about them ("How I learned to stop worrying and love reorgs," which appears earlier in the chapter). How does one stay sane while dealing with misinformed or apparently clueless new management? One of the more amusing pastimes is to imagine you are the new one in charge.

How would you change things if you became King of Windows? What would you do? Do you have the guts to stand up and be heard?

What's that? What about me? I thought you'd never ask…

Eric Aside I wrote this column shortly after my senior vice president, Jon DeVaan, got reassigned to run the core Windows division. I asked him if he wanted to review the column because people knew I worked for him. Jon said, "I think it would be best if I didn't edit you.… I am sure I. M. Wright would not listen to me if I tried :)." Since that time, Steven Sinofsky took over as president of Windows and switched it to a functional organizational structure like the one he had in Office. For Windows 7, Steven followed the basic outline of my plan, but he didn't bother with PUMs and applied much more control over individual teams' engineering processes. You certainly can't argue with his success, and I don't. Windows 7 is a wonderful product, notwithstanding the current competition with slate devices.

Have you any last request?

Here's what I'd do if I became King of Windows:

- Oh sure, I'd talk to everyone and all that. Yeah, yeah, yeah. Then…
  - I'd make the organizational structure match the architecture.
  - I'd determine the architecture based on the user experience.
  - I'd drive the user experience based on key scenarios in each usage category.
- But before selecting the key scenarios, I'd select my staff.
- My staff and I would analyze and establish the key scenarios.
- I'd assign staff members to lead the user-experience and architecture efforts.

- When these steps were completed, I'd set the organizational structure and assign leaders.
- I'd devote the first milestone of the new organization to quality and building out the engineering system defined in the architecture.
- I'd set feature priorities based on scenario priority and critical chain depth.
- I'd manage the release by minimizing feature completion cycle time in priority order.
- I'd demand that complete features meet or exceed the dogfood quality bar enforced by the engineering system.

Note that I don't mention any specific methods here, like Test-Driven Development or feature crews. While excellent methods like these should be supported and encouraged by the engineering system, they shouldn't be top-down edicts. Team dynamics should be driven from the bottom up.

Now let's break down each piece, since the details hold all the interest.

Prepare the ship

Talking to everyone and all that. This is more than a gratuitous gesture to the existing staff. Later I'll need to select my own staff, so I need to understand both the current situation and who I have available to serve. The key word here is "serve." I'm not looking for people to serve me or themselves. Either kind would poison my staff and our goals. I'd sooner see them fired. I'm looking for people to serve Microsoft and our customers.

> **Eric Aside** If there's anything I would change about management at Microsoft, or any company, it's the criteria used to judge who's available to serve. Yes, the best managers serve the company and our customers, but mixed in are managers who serve themselves, their superiors, or only their own organizations—and that is unconscionable.

Making the organizational structure match the architecture. You want the org chart to look like the architectural layout. This allows groups to operate with local control and independence as defined in the architecture. It also makes enforcing architectural boundaries easier. Naturally, you can't do this until you have an architecture. However, you can still think through what the structure would look like. I'd have product groups defined by the major components of the architecture. Each group would have a product unit manager (PUM), architect, group program manager (GPM), dev manager, and test manager.

> **Eric Aside** Drop the PUMs, but keep the GPM, dev manager, and test manager as a "triad," and you've got what Steven Sinofsky did for Windows 7.

Determining the architecture based on the user experience. This begs questions around timing and responsibility. Clearly, defining the Windows user experience and architecture must happen before any product groups are formed and any work begins; otherwise, you couldn’t set up the organization based on the architecture.

However, even after the experience and architecture are defined, they’ll be constantly reexamined, improved, and refactored. For this I’d have the user experience (UX) organization reporting directly to me under a UX director. I’d have a product-line architect with all product unit architects dotted-lined to her, and all of them meeting regularly as the Windows architecture team.

I’d also have directors of program management (PM), dev, and test reporting to me. They would be responsible for the engineering system and the long list of quality requirements like security, privacy, reliability, performance, maintainability, accessibility, globalization, and so on. The directors of PM, dev, and test would have all the product unit discipline managers dotted-lined to them. As such, my staff would drive engineering across the division.

Eric Aside Steven Sinofsky has the UX team report to the director of PM and the product-line architect report to the director of development. The GPMs, dev managers, and test managers report to their respective directors, and the directors report to Steven (no PUMs). It’s very simple and, dare I say, an improvement over what I proposed. Then again, he is King of Windows and I’m not.

Set a course

Driving the user experience based on key scenarios in each usage category. What makes a key scenario? It’s one that creates compelling customer value that differentiates the next version of Windows from all previous versions and all competitors. Naturally, we have marketing and technical research people to tell us what value would be compelling within each market segment. Heck, there’s no shortage of opinions on such things. However, my staff needs to choose our bet, what’s called our “value proposition,” because we are the ones who must believe in it and commit ourselves and all our resources.

Selecting my staff. Choosing my staff will be the most important thing I do. As I mentioned earlier, I’m looking for people whose whole focus is serving Microsoft and our customers, not me and not themselves. They may have career aspirations, but they figure if they serve Microsoft and our customers well, their careers will take care of themselves. As for their personal lives, I would expect my staff’s priorities to match my own: serving my family, friends, and community first, Microsoft and our customers second, and my own interests third.

Analyzing and establishing the key scenarios. Once my staff is in place, we start the hard work of defining the value proposition for the next version of Windows. This will mean long days filled with heated debates and detailed analysis. The result will be a clear vision for Windows and the centerpiece of our success.

Engage

Assigning staff members to lead the user experience and architecture efforts. With the value proposition in place, the work starts in earnest on the user experience and architecture. These teams should work closely together, informing each other of what's possible and what's impractical. At the end, we should have prototypes that key customers have validated and a product-line architecture that gives us confidence our goals can be achieved. The teams my UX director and product-line architect put together will form the core of their expanded teams once the organization is built.

Setting the organizational structure and assigning leaders. With the architecture defined, we can fill out the organization. My extended staff who helped create the value proposition, user experience, and architecture would have first choice at key leadership roles. After that, my staff would assign or acquire the rest. The full transition would be timed to occur a week or two after the Windows Vista ship date.

Devoting the first milestone of the new organization to quality, and building out the engineering system defined in the architecture. One of the critical pieces of the architecture is the definition of the engineering system—that is, the set of tools and processes that enable the engineering team to build and ship our products. Often a new architecture will require new technologies and new methods. Hopefully, we also apply lessons from past mistakes. The first milestone provides an opportunity to focus on unresolved quality concerns and building out improved tools, measures, and processes.

Eric Aside Many Microsoft product lines now devote the first division-wide milestone to quality and building out the engineering system, including Windows. That trend got started well before this column.

Navigation

Setting feature priorities based on scenario priority and critical chain depth. When the quality milestone is completed, the engineering staff will be ready to create new customer value. Creating that value in the right order is critical to shipping on schedule. I'd set the order based on scenario priority (do the most important features first) and critical chain depth (do the most critical dependencies first). This priority ordering and the way it was determined must be transparent to the entire engineering staff.

Managing the release by minimizing feature completion cycle time in priority order. How I choose to manage the release will be the third most important thing I do. The most important is choosing my staff; the second most is defining "done," which I'll describe next.

As for managing the release, I choose to minimize feature completion cycle time in priority order. In other words, I want features completed in priority order and in the shortest amount of time from when the feature team starts talking about the spec to when the feature is ready for dogfooding.

Eric Aside *Dogfooding* is the practice of using prerelease builds of products for your day-to-day work. It encourages teams to make products right from the start and provides early feedback on the products' value and usability.

Naturally, I'd use proven project management practices to track progress across Windows, but the division's ability to ship quickly and reliably is tied directly to each feature team's ability to do the same.

I will not specify how feature teams are structured or what methodology they follow. That is best determined by the teams themselves. But I will insist that we produce customer value efficiently, ensuring complete high-quality features get delivered as quickly as possible.

This will help everything we do, from early customer feedback to last-minute competitive responses. If lean methods like Test-Driven Development and feature crews happen to produce the fastest feature completion cycles times, all the better.

Demanding that complete features meet or exceed the dogfood quality bar enforced by the engineering system. The most important thing I do, besides choosing the right staff, will be demanding that features not be called "complete" until they meet a high quality bar. The quality bar I'd choose is that of being ready for dogfood. I'd enforce this quality bar within the engineering system and by always installing the latest dogfood build on my own machine. That way, my staff and I will keep a constant eye on whether or not the engineering system is applying the right metrics to maintain standards. Simply put, if your feature doesn't meet the quality bar, it can't and won't get checked into the main branch (or "main").

Eric Aside Feature crews typically work on branches of the source code from the main branch. This allows them to use the source control system to manage conflicts and builds, without impacting anyone beyond their small team. When the feature is fully developed and tested, ready for dogfood, the feature crew branch is merged and checked into the main branch. Of course, other feature crews check into main, so the longer a feature crew stays on a branch, the more difficult it is to merge at the end. That's another reason why you want to minimize feature completion cycle time.

Accountability

Having a bulletproof definition of complete is a key part of an overall philosophy of accountability without blame. Architectural boundaries are upheld because they match organizational boundaries. The right value, scenarios, and experiences are delivered because we enforce a priority ordering based on delivering that value. There is little wasted effort because we measure and reward those who deliver completed value the fastest. High quality is built in because you don't get credit for completing anything until it meets the bar.

There's no blame because the system, not an individual, enforces the requirement that quality, value, and a robust architecture are delivered efficiently. The only way to break the architecture, value, or quality is to clearly and transparently break the rules in collusion with your management. Maybe doing so is the right thing to do, maybe it isn't. Either way, the accountability is clear.

Windows, the next generation

There it is, 12 steps to a new Windows. Nothing too complicated or harsh—a few timing issues, perhaps, and some tool and measurement work to construct—but overall, I think it would be both doable and effective. Alas, I am not the king. At best, I could hold his bucket.

However, with change comes opportunity, and there's plenty of change afoot. Make your own voice heard. If you're not a king, send this to someone who is with your own thoughts and ideas. Who knows, we might even start a revolution.

Eric Aside The changes Steven Sinofsky put in place were difficult adjustments for many Windows engineers. However, the results speak for themselves. Windows has been far more efficient and innovative since Steven took over. It needs to be with the iPad and other slate devices becoming popular substitutes for consumer PCs. Personally, I don't think PCs are dead. Sure, some PCs may be multipurpose devices that fit in your pocket as phones and dock into a keyboard, mouse, and screen. Some may be slates that dock in a similar way. But some will remain in the nice laptop form factor. And I believe that five years from now, most will be running Windows regardless of their size or shape.

December 1, 2006: "Google: Serious threat or poor spelling?"

Perhaps I'm ignorant, but Google's attempt to compete with Microsoft is pathetic. Microsoft is far from perfect, but unless some company steps up and takes a real crack at us soon it will be tough to overtake us. That company certainly isn't Google.

> **Eric Aside** This was one of my most controversial columns. Many people inside and outside the company believe Google's approach to web-based services will do to Microsoft what Microsoft did to IBM's mainframe-based services. I agree with their points about the web changing the world and our legacy businesses (Windows and Office) making it difficult for us to be as agile as Google. However, I respectfully disagree with their conclusion.

Don't get me wrong, I'm quite impressed with a few things Google has done:

- They created a dominant Internet search engine.
- They created a powerful revenue model off a context-driven ad service with the help of two acquisitions.
- They created an engineer-friendly work environment to attract and retain strong talent.

> **Eric Aside** We can now add that they (Google) have created a developer-friendly phone and slate platform with cellular operator–friendly pricing.

Those are all great achievements.

However, those accomplishments aren't worth much in the long run. If Google is lucky, it will remain a successful niche player like Apple. If Google isn't quite so lucky and resourceful, it will join the ranks of Borland, Netscape, Corel, Digital, and others. Yeah, some of those companies are still in business, but they are shells of the competitors they once were.

They falter, we flourish

The foolishness and short-sightedness of our competitors never stops amazing me, and Google is no exception. It's almost too easy. Perhaps they think their collective incompetence will lull us into such arrogance, carelessness, and outright laughter that we'll forget who we are and what we do.

Eric Aside Okay, that was over the top. I wrote this when I felt the pendulum had swung too far in the direction of Google adulation and it was time to push back a bit. By the way, I have friends I admire at Google and other competitors. It's the business direction of these companies that I question, not their people.

We're certainly guilty of the arrogance, carelessness, and laughter. I'll be the first to admit it. But thank goodness we haven't forgotten who we are and what we do:

- We empower every person and device with software.
- We continually improve our products.
- We continually improve ourselves.

We follow these principles because we care about every customer. We want to change the world.

Failure by design

Given all the success Microsoft has had, you'd think our competitors would follow the same principles, but they don't. It's not that they fall short trying; it's that they specifically choose not to do it.

Most of our competitors stop improving their products. They move on, satisfied with the current version like we mistakenly did with Internet Explorer. That allows us to catch them and beat them by the time we get to our third version. Google is trying to correct this mistake right now. They've overemphasized new ideas to the point that their top executives are now telling engineers to pull back on creating new products and refocus on the ones they have.

Most of our competitors also stop improving themselves. We outspend all our rivals by a wide margin in research and development. But as old competitors fall behind, new ones take their place.

In addition, we've got a long way to go before we meet our customers' higher expectations for mature product quality. Lucky for us, while Google's practices are sound, they aren't explicitly focusing on quality engineering. Their biggest challenge to us outside of their excellent yet mostly stagnant products is their engineer-friendly work environment. In time, our dedication to continual improvement will beat them in both areas.

Eric Aside While Google has incrementally improved their search, ads, browser, and Android OS, they haven't yet shown they can make revolutionary improvements in customer experience. It took years for Microsoft to move away from developer UIs and feature flood and move toward integrated, delightful customer experiences. Customer experience is where Apple excels. Google is far behind, while Microsoft has gradually been catching Apple and moving ahead in our newest releases. (Apple zealots can now stare lovingly at their iPhones and reassure themselves by counting their rosary button.)

Smart people, smart clients

But let’s say Google gets its act together and starts continually improving its products and itself. Google has done an excellent job of attracting and retaining strong engineers. They could decide to refocus their efforts on customer value, product quality, integration of services, and multirelease plans. Wouldn’t they be a threat, especially with their fantastic ad platform that practically prints money?

No. At best, Google will be a niche player like Apple. Why? Because of our first principle—which Google, like Apple—has chosen to disregard: “We empower every person and device with software.”

Apple wants to empower only the people and devices they choose. The strategy has its advantages, but it structurally limits their market. Too bad, they’d be a tough competitor otherwise.

Eric Aside I wrote my dissertation on a Mac. I love Apple’s dedication to design and user experience. Over time, dedication to quality experiences, along with quality engineering, has been the growing source of our own success in old and new areas.

Google wants to empower every person with software, but not every device. They’d rather keep the smarts on the server or in generic snippets on the client. It’s the old “network computer and dumb client” story. It’s remarkable how many times this losing strategy has claimed victory over the smart client.

Eric Aside With Android, Google has moved onto phones and slates. This is great, but they are using these devices as a platform for their ads. Their business model hasn’t changed. They charge operators little or nothing for Android and make their money on the services connection and advertising. I’m not saying that’s evil—I’m saying that’s limited. Google isn’t on devices to make the most of the devices. They are on devices to make the most of their ads. In the end, their subpar customer experience will place them behind their competitors. Wait another few years and you’ll see what I mean.

Microsoft brings the power of software into every person’s office, home, and hands. The closer you can bring that power to people, the more extraordinary and valuable their experience becomes. Google will lose as long as they limit their reach. How badly they lose depends on whether or not they can manage to at least hold onto a few server-based scenarios.

Eric Aside Some might argue that AJAX provides enough smarts for a client, or that Google has been shipping smart clients, like Google Earth (which I love). However, AJAX is limiting, and Google’s software development process, which many consider its strength, is optimized for web delivery. Google would need to make big changes to focus on smart clients, much like the changes Microsoft has already welcomed to focus on web services.

Staying vigilant

Just because Google has doomed themselves, that doesn't get us off the hook. We have to keep improving ourselves, creating an environment that attracts and retains great people, including Google employees when they abandon ship. We have to keep improving our products and services, providing more compelling value than Google with integrated, exceptional experiences.

We must stay vigilant because some day—perhaps starting now in someone's garage, perhaps in India or China—there will be a competitor who understands our principles. They won't have our baggage of legacy code and legacy engineering methods. They will focus on lean, high-quality practices and products, starting small and slowly expanding their influence. It's happened in every industry, and it can happen to us.

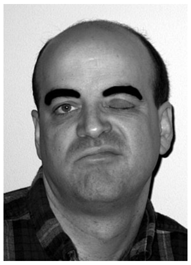
Staying out in front

What's different is that we have sensed this change in advance. We know the market has matured. We know we must transform our engineering systems and approach in order to produce higher quality products and services that provide more value with greater predictability and efficiency. Our principles of continually improving our products and ourselves are slowly making that transformation a reality. It takes time and patience, but we're really good at taking the time and being patient.

We know in the end we'll win. You can be a part of this exciting time. You can dedicate yourself to improving our engineering and our products. You can put quality and customer value first, for everyone on every device.

Change is difficult. There are always those who prefer the status quo, especially when it comes to old habits. But with change comes opportunity, opportunity for you to define and even lead the new status quo. Grab it while clueless competitors like Google are giving us the chance.

April 1, 2007: “Mid-life crisis”



Recently, I put a significant deposit down on a sports car—a two-seat Tesla Roadster convertible that goes from 0 to 60 in four seconds. It’s an electric vehicle that travels over 200 miles on a 3.5 hour charge. A couple of months from now, I’m taking my sabbatical to travel abroad. I know what you are thinking. You’re guessing I’m in my mid-forties.

> **Eric Aside** You can read more about the Tesla Roadster at *www.teslamotors.com/*. I took delivery in 2009 (model year 2008). I’ve now driven the car every day to work for the past two years. It’s the easiest, most economical, most environmentally sustainable, and most fun car in the world to drive. It also requires almost no maintenance. You can read more about my experience on the blog I wrote for Tesla: *www.teslamotors.com/blog/we-didnt-want-roadster*.

Yes, I’m having a mid-life crisis, or as I like to call it, the time of my life. Still, my wife and I are happy together; I love my kids and have no intention of leaving them; my job is great and I’m as passionate as ever about Microsoft. I’m just preparing to travel the world and buying a hot electric sports car. What gives, and more importantly, why should you care?

What gives is that my situation has changed, providing new opportunities and new pitfalls. Why you should care is that Microsoft is in the same situation.

> **Eric Aside** I also used my sabbatical to assemble the first edition of this book. At Microsoft, your vice president can award you a sabbatical after a certain number of years of consistently excellent contributions to the company.

You’ve changed

Up to this point in my life, a sports car was unthinkable. Before I had kids, I couldn’t afford such a thing. Once I had kids, a two-seat roadster wasn’t practical. Now my kids have grown, as well as my savings, so for the first time a sports car is feasible.

Of course, just because it’s feasible doesn’t mean I want a sports car. But this is an electric, blow-your-doors-off, twist-your-neck-loose car. Such a car didn’t even exist till last year. The technology has improved, the business climate among other things has changed, and I put down a deposit.

Likewise, before now, taking a sabbatical didn’t make sense. It took me a while to earn the sabbatical. Once I did, between my kids and my job, taking that much time off was unthinkable. Now, my kids don’t need me as much and I’ve grown my staff to the point where they can cover for me. So taking the sabbatical is viable.

Microsoft has grown up too. We used to be struggling to establish ourselves. Even once we were successful, we had to work diligently for years to be taken seriously. Now we hold a leadership position, have money in the bank, and find ourselves at a crossroads.

Just another tricky day

So is this a crisis? Not for my personal life. My job and relationships are stable. My finances aren't in jeopardy. There are people willing and able to cover for me. All this "crisis" amounts to is an opportunity to indulge myself.

What about Microsoft? People describe our situation as a crisis. They say we've forgotten how to ship; that we are bogged down in process "taxes"; that our approach to managing people and projects is antiquated; and that our competitors are more nimble and relevant in today's market for talent, products, and services. They are right. What went wrong? We're crawling when we should be cruising.

Eric Aside I don't think it's as bad as some critics say, but their concerns are valid.

How bad is Microsoft's mid-life crisis? I'd say it's worse than mine, but better than the one an old officemate went through. He divorced his wife, left his kids, and remarried a woman half his age. IBM's mid-life crisis was kind of like my officemate's mid-life crisis. Apple's mid-life crisis involved two divorces before returning to their first spouse. While both IBM and Apple pulled through, they definitely suffered.

How does Microsoft avoid suffering through this time of transition? We don't avoid it; and, as it turns out, we haven't. We are suffering, and we'll continue to suffer. Transitions are hard. However, there are things we can do to minimize the pain and maximize enjoying the opportunities that change brings us.

Leave little to chance

Let's take the conversation back to me. Remember, I'm the one taking a long vacation and then returning to a cool electric sports car and the family and job I love. Why isn't my wife leaving me? Why are people willing and able to cover for me? I've had my share of good fortune, no doubt, but my situation isn't an accident, nor is it assumed.

Had I bought a speedy two-seater years ago, my wife probably would have tolerated it, but it would have caused my family grief. The costs to buy, insure, and maintain it would have put stress on our finances. I wouldn't have covered half the driving duties after work because the kids couldn't safely sit in the front seat. What's worse, I would have missed opportunities to connect with my kids.

But I didn't buy the sports car earlier. As a result, my relationships with my wife and kids are stronger, and our finances are more secure. Thus, getting one now isn't a big deal. As an added bonus, new technology has provided an electric alternative.

The moral: Don't prematurely commit to exciting new areas just because they're there—make sure you establish your dependents and can manage the risks. You might even find that technology improvements by then are far more favorable.

Eric Aside By the way, this is no judgment on people with sports cars. Each person's family and job situation is different. This is more about how a certain approach has helped my family, how it has helped my job, how I successfully weather the changes life has presented, and how that approach could help Microsoft do the same.

I don't think the boy can handle it

Had I taken my sabbatical when it was awarded years ago, I would have compromised my family, my team, and even the quality of my vacation. The family aspect is simple. Today, my kids are older and more independent. This means less work for my wife and less guidance needed from me. As an added bonus, we have more flexibility for where and how long we travel.

The impact on work is even greater. Years ago, my team was new. I hadn't had the time to hire the right people and develop them sufficiently to step into my role. I also hadn't established the practices and guidelines we needed to provide reliable results. Today, while the situation isn't perfect, I've got a strong, experienced team behind me who are eager and able to cover my duties.

The moral: When you invest in developing good people, practices, and guidelines, it gives you the freedom to explore new areas without worry or compromise, while giving your people the chance to continue your work and make it their own.

Not getting any younger

By now, many of my readers are seeing how this relates to Microsoft, but in case you are living in a state of blissful ignorance or denial, allow me to elaborate.

Microsoft is entering middle age:

- **Our market has matured.** We are taken more seriously by customers, partners, and governments. This is good for us, but it also raises expectations for what we deliver and how and when we deliver it. We can't afford to be careless or impetuous.

- **We carry around tons of legacy baggage.** Without the extra weight, our competitors can be more nimble, taking advantage of new technology and revenue models faster than we can.
- **Our senior management, middle management, and college recruits all come from different generations.** This generation gap is quite serious. Senior management never had to ship software with a thousand engineers. Middle management doesn't know agile from fragile. And college recruits haven't a clue about production-quality, world-ready code. This leads to conflicting expectations of development time and engineering approach, as well as to poor decisions based on misinformation.

Don't panic

What do we do? First off, don't panic. We are rich, famous, and darn good looking (from a portfolio perspective). There are worse situations.

Second, remember we're not a human being. We're a company. We only get old and decrepit if we get set in our ways, preventing innovation and keeping the next generation from taking our place. We can learn and renew ourselves if we just choose to do so.

Applying the first moral, we must establish our dependents and manage our risks before we commit. We're too old to be screwing around with unstable base components, like we did with Windows Vista. Aggressively investigate new technologies and innovate our products, but don't commit till dependencies are solid and the risks can be mitigated.

Eric Aside We ended up cutting back some new technologies in Windows Vista because they weren't stable enough to ship with high quality. That's more than a shame. It cost us time trying to put them in, and it cost us more time removing them later.

Applying the second moral, we must invest in developing good people, practices, and guidelines. We're too old to be devastating and demoralizing our staff with death marches (which you can read more about in "Marching to death" in Chapter 1). We must be patient, allowing folks to learn and grow on their own, while teaching them the lessons we've already experienced. This means letting go and trusting our next generation to prove themselves, but having the right guidelines in place to prevent a catastrophe.

Nobody's perfect

We've been less than perfect in establishing our dependencies, managing our risks, developing our people, and having the right practices and guidelines in place. Transitions are hard, and there's no getting around it.

What helps is maturity at all levels. In other words, acting our age:

- No responsible adult drives around in a car with bald tires or builds a house on a fault line. That's foolish. We need to stabilize our dependencies before we depend on them. We can't compromise on quality.
- No decent parent still tells their grown kids what to do. That's childish. Management needs to provide engineers with knowledge, experience, and guidelines; clearly explain the expected results; supply feedback that warns of danger; and then trust engineers to make it happen however those engineers see fit.

Taking these simple steps will do more than make us act our age. We'll develop new leaders to step up when the old generation steps down. While the younger generation continues our work and makes it their own, we can explore new areas without worry or compromise.

Getting older as a company doesn't mean getting old. We stay young and agile by growing, protecting, and trusting our youth to keep us young. I can't wait to take delivery of that electric sports car.

Eric Aside If current actions are any indication, we have learned many lessons well. We've avoided telling engineers how to work; instead, we have focused on the results we expect. We've changed our approach to dependencies and shipping, making ourselves more lean, agile, and reliable. Sure, Microsoft still has work to do, and I love being part of the solution.

November 1, 2008: "NIHilism and other innovation poison"

Is innovation the act of creating something new (as the dictionary claims) or is it building upon the work of others? To me this is a fundamental question that Microsoft as a company and as a culture has gotten horribly wrong. We deal with the consequences every day. It shakes our self-esteem and cripples our ability to innovate.

The right answer is that innovation is enhancing the work of others—nothing is new. Consider what people claim to be innovative—the iPhone, hybrid cars, and Facebook. Are those new? Are you kidding? Not only would those advances have been impossible without the foundational technologies behind them, but almost every aspect of those innovations had been created earlier in some form. The innovation was putting old technologies together into compelling new experiences and successfully marketing them to a broad audience.

This isn't an indictment of innovation or great advances like Facebook. Quite the opposite—this is a call to acknowledge that innovation is enhancing the work of others. Microsoft culture, in an act of misguided self-mutilation, has connected being innovative with creating something "new" on your own. This has directly led to three crippling consequences:

- We doubt our innovative spirit as a company.
- We create disconnected new features instead of compelling new experiences.
- We reject work we didn't create ourselves as a company, team, or individuals, which impairs our productivity, our intelligence, and, ironically, our ability to innovate (an advanced, tragic case of *Not Invented Here*, also known as NIH).

Why the long face?

We doubt our innovative spirit as a company because much of our work goes into enhancing existing products and services, which in turn were built upon ideas from a wide variety of sources. Microsoft haters out there claim this proves Microsoft isn't innovative, and we accept that guilt. It sickens me.

Is Apple not innovative because it "borrowed" from Xerox PARC (Palo Alto Research Center)? Is Google not innovative because it enhanced Yahoo? Is Toyota not innovative because it built upon Henry Ford's ideas of the automobile and Lean Manufacturing? That's both stupid and insulting.

Being innovative is all about enhancing the work of others. We enhance our own great work and build upon the great ideas we find around the world in all fields. Yet our self-esteem is shaken, even while every new release is filled with innovation.

Perhaps the problem is that our innovation isn't as compelling as the iPhone. Why?

Compelling, and rich

We are so eager to promote our own individual new, small ideas that we fail to create compelling new experiences. You can't market a plethora of disconnected features to a broad audience, so our individual innovations are left unnoticed.

That's beginning to change as groups realize that customer value comes from designing compelling end-to-end experiences. More and more Microsoft products and services have a consistent feel and a complete story. It started with Hardware and Office, crystallized in Xbox 360, and is making its way into Windows—Windows Vista, Windows Server, Windows Live, and Windows Mobile. We don't always get it all right, nor do any of our competitors, but each new release shows progress.

Tragically, our company's culture of individual innovation works against creating compelling end-to-end experiences. You can't assemble seamless scenarios without selflessly supporting the salient story. You have less unbridled innovation from the individual engineer, but the result is greater innovation from the customer's perspective.

As any artist knows, constraints often lead to tremendous creativity. However, individual engineers must first let go of their own preconceptions and embrace the constraints of the larger composition before they can innovate in concert, like a symphony instead of a soloist.

Our greatest successes both monetarily and in emotional connection came from working together over several years to get the experience right for customers. Some of the ideas may have come overnight, but as Thomas Edison said, "Genius is 1 percent inspiration and 99 percent perspiration." Those innovative engineers from groups like Office, Developer Tools, and SQL Server, with the patience to iterate, focus on the customer, and bring full experiences together, are the ones who receive the greatest rewards. Yet Microsoft engineers still cling to the notion that working independently from nothing is the secret to success.

I'll do it myself!

Unfortunately, Microsoft's internal mythos is all about the individual hero rewriting the kernel memory protection scheme over a sleepless weekend. We value independence, passion, and boldness over compliance and conformity. I love that about Microsoft, but it needs an update.

Eric Aside A reader pointed out that Microsoft's values of independence, passion, and boldness are how we see ourselves within our engineering teams, not how the company projects itself to customers. That's an interesting point and a potential future column.

The problem is that independence, passion, and boldness somehow became "do it yourself." If you didn't invent it, it's not what you need. If your team didn't invent it, it will only cause trouble. If Microsoft didn't invent it, it's dubious at best. That NIH worldview is not only wrong and shortsighted, but it slows our success and paralyzes our progress.

You can be independent, passionate, and bold while responsibly meeting requirements and enhancing a desired customer experience. You can be independent, passionate, and bold from the advantageous platform of others' foundational work.

Conversely, if you choose to go it alone you suffer greatly:

- Your productivity drops.
- Your intelligence drops.
- Your innovation drops.

Maybe I could turn this thing into my advantage

When you reject work that was NIH, you lose the advantage of time, effort, bug-fixes, and knowledge that went into that work. Would you write an operating system from scratch to optimize a small application? Of course not. By using an existing operating system you save incredible effort.

The same is true of using existing libraries, services, tools, and methods. The key to gaining advantage instead of hardship from using the work of others is ensuring that the things you depend on are stable. Take a dependency on version n-1, be careful about using the latest and greatest. That goes for libraries, tools, and even techniques. Grief is avoidable.

Eric Aside When using existing libraries, services, tools, and methods from outside Microsoft, we must be respectful of licenses, copyrights, and patents. Generally, you want to carefully research licenses and copyrights (your contact in Legal and Corporate Affairs can help), and never search, view, or speculate about patents. I was confused by this guidance till I wrote and reviewed one of my own patents. The legal claims section—the only section that counts—was indecipherable by anyone but a patent attorney. Ignorance is bliss and strongly recommended when it comes to patents.

I'm good enough, I'm smart enough

Rejecting work NIH also leads to brain decay. At Microsoft, we hire the brightest engineers available. Are they still the brightest five years later? Well, that depends. Did they learn something new every year, or did they rebuild the same thing over and over again?

If you have to reinvent a build system every time you switch teams, are you smarter? If you have to reinvent a collection class, a threading framework, or a test harness every time you switch teams, are you smarter? If you have to reinvent project management techniques, spec templates, and bug field definitions every time you switch teams, are you smarter?

No, you're not smarter. You are getting dumber. Congratulations. You don't get smarter by repeating yourself or by repeating others and their mistakes. You get smarter by building upon the knowledge of those before you.

Those who do not learn from history

Ignoring what others have learned and gained before you, both good and bad, puts you at an innovation disadvantage. As I already mentioned, there's the productivity disadvantage—going it alone is never as fast as building upon the progress of others. However, there's also a significant contextual disadvantage.

If you ignore past work, you're liable to make the same mistakes that people made in the past—avoidable mistakes that cost you time. Sure, you might want to revisit those challenges. Perhaps there's an opportunity for a breakthrough. But do so knowingly, not like a dolt.

If you ignore past work you're also liable to unknowingly duplicate what others have already achieved. You'll feel smart and proud right up to the point that you look like an idiot. Meanwhile, you've lost time and accomplished nothing new. NIH results in nihilism. Fitting, I suppose.

Eric Aside This column was widely read, and a number of people misinterpreted it to advocate a passive rather than active approach to innovation. Just because you are enhancing the work of others doesn't mean it will come to you easily and no risk is involved. Plenty of vision, sweat, and gambles go into realizing groundbreaking ideas. So does plenty of study of current technology, markets, and the deep needs of customers. You aren't alone, but you also can't expect others to do all the work for you.

If not me, who? And if not now, when?

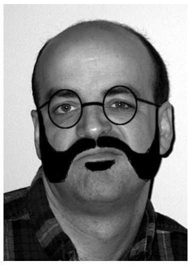
Microsoft is filled with smart, passionate, and innovative people. We are an innovative company, in spite of ourselves at times. We need to stop thinking of innovation as an individual effort that appears miraculously from the void, and start thinking of innovation as it really is—a culmination of effort focused on thrilling the customer.

Where do your individual ideas fit? They fit within the context of the larger desired customer experience, and they build upon the learning we've gained as a group. The result is groundbreaking innovation that expands the state of the art and excites the hearts and minds of our customers.

So when you're going to work on a new team, project, or great new idea, understand how it makes the customer experience we've set out more compelling; and learn what others have discovered about your work. Note that I said "what others have discovered" not "if others have discovered." There is nothing new. Someone out there has tried something similar and will have work or knowledge you can use. You can ignore that and be slower, dumber, and less accomplished. Or you can take it to the next level and create real innovation.

Eric Aside If you are looking to get ahead and build upon the great work of others, one of the best places is in the shared source community (internally at CodeBox and externally at CodePlex). By using personal branches, you can customize and enhance code and tools all you like, while still updating with the latest and greatest improvements from others. By submitting your changes to the main line, you advance the state of the art and might even enhance your image, too.

February 1, 2010: "Are we functional?"



When Steven Sinofsky and Jon DeVaan took on joint management of Windows 7, they made several significant changes to the entire organization. Two profound changes were creating a single centralized plan and switching to a functional organizational structure. Given the success of Windows 7, some Microsoft engineers are asking, "If my PUM is a bum—is it time to throw the bums out?"

Maybe, but let's not get too hasty. Knee-jerk nitwits who act before they think are doomed to repeat old failures in new ways. Everything has its own purpose and place. Before you throw the bums out, consider the role that product unit managers (PUMs) play and how best to balance product and functional demands. Forgotten? Never understood them in the first place? I'll remind you.

Eric Aside "Is your PUM a bum?" was one of the most popular of my early columns. It appears earlier in this chapter. PUMs are typically responsible for a self-contained collection of functionality, such as Microsoft Excel, DirectX, or ActiveSync, though none of those groups are run by PUMs anymore.

I'm PUMed

PUMs are multidisciplinary managers responsible for the strategic and operational decisions needed to engineer a Microsoft product or major component—at least in theory. In practice, PUMs are often dysfunctional—marginalized from above by their general manager (GM) or from below by their discipline managers. However, when used properly, PUMs give small innovative products agility in their market by handling business and execution decisions like a startup-sized team.

PUMs typically appear toward the bottom of product organizations and don't appear at all in functional organizations. Functional and product organizational structures aren't new—modern versions of both have been around since the 1970s. Is there a "right" choice? Let's break it down.

What's your function?

Functional organizations, like Windows now, are organized by disciplines. The division president has a director of each discipline reporting to her (such as a director of development or a director of testing). There are no multidisciplinary managers within the division.

Product organizations, like Windows Server, are organized by products. The division VP has a set of GMs reporting to him who each own a set of related products. Each GM has a set of

PUMs who in turn own a specific product. Each PUM has discipline managers for each discipline reporting to her. Basically, a product organization becomes a functional organization at the PUM level.

In either case, to get the product built, engineers need to work across disciplines, typically in feature teams. Thus, both organizational variations are "matrix organizations"—people working together across functional boundaries (silos).

Based on this description, you could and should be thinking, "Wait, every company I know starts at the top as a product organization and switches into a functional organization somewhere down the chain." Exactly. Even new-age, hip, cool, out-there companies start at the top as product organizations and switch to functional organizations at the bottom. The only issue is when you switch. Windows now switches at the division president level, Office switches at the GM level, and Windows Server switches at the PUM level.

Eric Aside Small teams often have people playing multiple discipline roles, so they may not seem functional at the bottom. Yes, I meant that.

Self-directed teams that choose to forgo specialization and have everyone do everything are the closest to being product organizations throughout.

Make the switch

At what level should an organization switch from a product structure to a functional structure? Idealistic neophytes would say companies should be flat—functional from the top, or, better yet, have a product structure with one boss and interchangeable people underneath, like a crowd-sourced project. Admittedly, that's a beautiful dream, where people share the wealth, award each other, and live in peace, harmony, and prosperity forever. Naturally, it's not sustainable.

The problem is a lack of alignment around business direction. Once you get to projects that require orchestrating more than 150 people, you must create structural mechanisms to keep everyone headed toward the same goals and timelines. Some companies, like Gore (makers of Gore-Tex), are specifically designed around not building anything that requires more than 150 people. However, you need large groups to build the integrated capability of an operating system (OS), a suite of applications, or an end-to-end set of services.

Every product needs a clear business direction, a vision, which lays out the goals for the upcoming release (themes and scenarios), the basic principles behind it (tenets), and the timeframe by which it must be delivered (high-level schedule). It also helps to have a shared understanding of how this direction was reached and where it hopes to go. Ideally, everyone in the organization should be part of forming this direction, but one person owns it for both coherence and accountability. That person is the business owner.

Below the business owner, you want everyone to follow the vision. You don't want more business owners—you want people who can align the organization around executing the plan. At what level should an organization switch from a product structure to a functional structure? At the business owner. Windows is a single business—we don't sell the product in pieces. It makes sense to be functional below Sinofsky. Office sells its products in several pieces. It makes sense to have GMs own those pieces and be functional below the GMs.

Eric Aside Internet Explorer does ship separately from Windows. Guess what? It has a GM.

You can go your own way

Why not have sub-business owners below the business owner? Because you don't need multiple business plans—you want a single vision. Business owners can't help themselves. They want to set direction, to make plans, to make business decisions. After all, they've got no other responsibility besides pure people management.

Eric Aside There are several other advantages to being a functional organization below the business owner.

- ❏ Your discipline leaders don't have to abandon their discipline and become a PUM to continue to grow. This means your most senior engineering people are solving engineering problems instead of people, administration, or business problems.
- ❏ Your organization tends to flatten without the extra layer of multidisciplinary management. This means fewer managers and less reinterpretation and misinterpretation of messages top-down and bottom-up.
- ❏ Your organization tends to have a more consistent structure, creating clearer roles and responsibilities and making it easier for people to work peer-to-peer instead of through escalation (you know who your peers are and what roles they play).
- ❏ You have more natural and recognizable mentors for your engineering staff because your senior engineering leaders stay in discipline.
- ❏ You can more easily share practices, code, and tools across the organization (no more fiefdoms).

Does this mean there's no room for innovation and autonomy under a single vision for a product? Oh please! I hear this whine so much it makes me ill. A product vision is not a feature spec or a design document. If it were, the entire Microsoft engineering staff would be about 125 people and a ton of vendors.

There are millions of ways to achieve the themes and scenarios laid out in the vision that meet the tenets and the schedule. The vision provides intent—there's plenty of room for inspiration. Perhaps a scenario can be met with a new approach, removing complexity and time. Perhaps a tenet can be achieved in ways that save billions of dollars over the next few years. Or perhaps there’s just a simple change based on customer feedback that will delight a whole category of users—a change that others missed or didn’t have the courage to champion.

It’s true that functional organizations share one business plan, so large functional organizations can’t radically change their business plans overnight. That’s prudent for a large product like Windows, but may not make sense for rapidly changing areas. You pick the model that best fits your business and market.

Eric Aside People might be screaming, “Yeah, what about slate devices and app stores? Windows is being left behind because it’s so slow to respond!” In business you can chase or you can lead. Microsoft went through a chasing phase—now we mostly lead. Sometimes we get caught flatfooted by terrific innovation from our competitors, like the app store and touch-first displays from Apple or the natural Wii controller from Nintendo. But we’ve learned not to have quick and thoughtless reactions. Instead, we study the problem, deeply understand the larger opportunity, and respond with significant innovations of our own, like the marvelous Windows Phone, which has vastly simplified my daily life, and Kinect, which has completely transformed the natural user interface. You can expect Microsoft to continue this more thoughtful approach going forward.

The one thing you do have at Microsoft, regardless of your organization’s structure, is the autonomy to make your own choices and guide your own work. Not everyone takes full advantage of that autonomy. People confuse alignment and constraints with shackles. As I’ve said before, artists, architects, and creative minds do their best work when provided constraints. It’s the challenge of building something new within the confines of the existing world.

Eric Aside Individual autonomy has gotten Microsoft into trouble over the years. It makes us inefficient at times. It nurtures a not-invented-here culture in which everyone is reinventing the same kinds of wheels because people have too little incentive to use each other’s work. However, on balance I love the autonomy. That trust coupled with Microsoft’s desire to change the world through software is what makes Microsoft unique.

Finding the right mix

To run successful large projects, you need leadership and decision making around both product and function. The balance determines success. Too much function focus and you lack a cohesive product. Too much product focus and you lack consistency, efficiency, and resource flexibility.

An approach that works well for engineering organizations is to switch from a product structure to a functional structure at the level where product direction is set and aligns the functional organization. Division leadership should determine that level based on their business strategy.

PUMs might make sense for the Windows organization if we sold Windows in lots of pieces. However, Windows would cease to be the integrated platform for thousands of solutions our partners create and our customers rely upon. So we sell Windows as a single platform, and Windows has a functional organizational structure.

Should your group adopt the Windows model? That depends on your business and your market. Who should be accountable for business decisions? At what level should those decisions be made? You need to match your organizational structure to your business strategy. If yours doesn't match, perhaps it is time to throw the bums out.

Eric Aside Microsoft is far from perfect. We still have plenty of work to do, and I love being part of the solution. I also love that I was trusted to represent Microsoft fairly and honestly when publishing this book. Thanks for taking the time to read it; I hope it was both enjoyable and helpful.

Glossary

2.5, 3.0, 3.5, 4.0, 4.5 (also known as ratings, the review system, the curve) Microsoft's old rating system, which was changed in the spring of 2006. Ratings of 2.5 and 3.0 were undesirable. Ratings of 4.0 and 4.5 were highly desirable. A 3.5 rating was readily accepted and the most common.

BillG, or Bill Bill Gates, Chairman of the Board of Microsoft.

black box testing Testing that treats the product as a black box. You don't have any access to or knowledge of the internal workings, so you probe it like a customer would—you use it and abuse it till it breaks.

Bohrbug A predictable and repeatable software issue (bug). It contrasts with an unpredictable and seemingly random Heisenbug. The terms *Bohrbug* and *Heisenbug* date back to Jim Gray's 1985 paper "Why Do Computers Stop and What Can Be Done About It?".

BrianV, or Brian Valentine Brian Valentine, former Microsoft senior vice president of the core Windows division.

buddy drop (also known as private build or buddy build) A private build of a product used to verify code changes before they are checked into the main code base.

bug, or work item Internally, we use the term *bug* to refer to anything we want to add, delete, or change about a product, what most people generally call a *work item*. Naturally, this includes code mistakes, the more traditional kind of "bug."

Build Verification Test (BVT) Checks whether a software build satisfies a set of requirements.

calibration A process Microsoft uses as part of its differentiated pay system to calibrate expected and exceptional contributions of a group of employees in comparable careers stages and roles.

CareerCompass A Microsoft internal web application that allows employees to assess their competencies and skills against standards for their career stage.

Career Stage Profile (CSP) Detailed descriptions of the work expected of employees at different career stages for different disciplines. CSPs also outline individual contributor and manager growth paths.

CodeBox (also Toolbox or CodePlex) A repository for shared tools and code. CodeBox is an internal code-sharing repository. Toolbox is an internal repository, mostly focused on tools and scripts. CodePlex is an external code-sharing repository.

code complete The stage at which the developer believes all the code necessary to implement a feature has been checked into source control. Often this is a judgment call, but on better teams it's actually measured based on quality criteria (at which point it's often called *feature complete*).

dogfooding (also known as "eating your own dogfood") The practice of using prerelease builds of products for day-to-day work. It encourages teams to make products correctly from the start, and it provides early feedback on the products' value and usability.

external bug, or external A bug in code not owned by the team. These bugs should never be ignored unless there is a straightforward workaround.

feature A self-contained collection of functionality needed to provide incremental value to a product. Although features can be large, ideally a feature is work broken down to the point of requiring no more than five weeks of effort to design, develop, and test.

feature crew A small, cross-discipline team tasked with a single feature (or closely related small features) to design, spec, develop, and test together from start to finish. Feature crews are typically virtual teams: the team members don't all report to the same manager.

Heisenbug An unpredictable and seemingly random software issue (bug). It contrasts with a predictable and repeatable Bohrbug. The terms *Bohrbug* and *Heisenbug* date back to Jim Gray's 1985 paper "Why Do Computers Stop and What Can Be Done About It?".

informational An informational meeting you can request with a hiring manager before you apply for an open position.

milestone Project dates that organizations (from 50 through 5,000 people) use to synchronize their work and review project plans. The term *milestone* is also used to refer to the work time between milestone dates. Milestone durations vary from team to team and product to product. Typically, they range from 6 to 12 weeks each. Calling a milestone a *sprint* does disservice to both terms.

PREfast, or Code Analysis for C/C++ PREfast is a static analysis tool for the C and C++ programming languages that identifies suspect coding patterns that might lead to buffer overruns or other serious programming errors. Though initially used only internally, it shipped as part of Visual Studio 2005.

product unit manager (also known as PUM, Group Manager, Director) The first level of multidisciplinary management. Typically, the PUM is in charge of a self-contained collection of functionality, such as Excel, DirectX, or ActiveSync.

program management, or program manager (PM) The engineering discipline primarily responsible for specifying the end-user experience, including the overall project schedule, which determines when that experience will release.

project, or release The entire collection of work necessary to release a specific version or service pack of a product.

RAID (related terms include Product Studio, bug database, work item database) RAID is a database and client for tracking work items, which can include feature work, bug reports, and design change requests.

reorg Short for *reorganization*. Typically, a reorg starts at the top and works its way down over a period of 9 to 18 months.

RDQ, or PSQ A work-item database query used to determine the state of work for a project.

scenario A description of an end user accomplishing a task that may or may not be implemented in the current product. Scenarios typically involve using multiple features.

software development engineer (SDE) A software developer. This refers to the people who write the code and construct the customer experience.

Source Depot, or source control Our large-scale source control system that manages hundreds of millions of lines of source code and tools, including version control and branching.

specification (spec) Documentation that specifies how a product should be experienced, constructed, tested, or deployed.

SQM (also known as Software Quality Metrics, Customer Experience Improvement Program) SQM is the internal name for the technology behind customer experience improvement programs for MSN, Office, Windows, and other applications. These programs anonymously aggregate customer usage patterns and experiences. (Please join when you install our software; it lets us know what works and what doesn't.)

SteveB, or Steve Steve Balmer, the Chief Executive Officer (CEO) of Microsoft.

STRIDE A mnemonic device to help people remember the different kinds of security threats: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege. *Writing Secure Code* by Michael Howard and David LeBlanc (Microsoft Press, 2002) has all the details.

Test-Driven Development (TDD) An Agile methodology in which developers write tests for code before the code is written.

Toolbox (also CodeBox or CodePlex) A repository for shared tools and code. Toolbox is an internal repository, mostly focused on tools and scripts, not code. CodeBox is an internal code-sharing repository. CodePlex is an external code-sharing repository.

triage (also known as bug triage or issue management) A regular meeting held toward the end of development cycles to manage issues. Typically, these meetings are attended by representatives from the three primary engineering disciplines: program management, development, and test.

Trustworthy Computing (TwC) The Microsoft initiative on security, privacy, reliability, and sound business practices.

User Experience (UX) Refers to the User Experience discipline, which includes mostly designers and usability experts.

Watson (also known as Crash Watson or Windows Error Reporting) Watson is the internal name for the functionality behind the Send Error Report dialog box you see when an application running on Windows crashes. (Always send it; we truly do pay attention.)

Watson bucket Each "Watson bucket" represents and stores a customer issue that thousands, sometimes millions, of customers have experienced. Engineers inside and outside Microsoft can query which buckets came from issues in their software.

white box testing Testing that uses instrumentation to automatically and systematically test every aspect of the product. Microsoft is steadily replacing its black box testing with white box testing.

zero bug bounce (ZBB) The first moment in a project when all features are complete and every work item is resolved. This moment rarely lasts very long. Often within an hour, a new issue arises through extended system testing, and the team goes back to work. Nevertheless, ZBB means the end is predictably within sight.

Index

About the Author

Eric Brechner has been at various times a development lead, development manager, development director, and director of engineering learning and development for Microsoft Corporation. He has worked on Image Composer, Office, Office.com, and Xbox.com. Before joining Microsoft in 1995, Eric was a senior principal scientist at The Boeing Company, where he worked in the areas of large-scale visualization, computational geometry, network communications, data-flow languages, and software integration. He was the principal architect of FlyThru, the walkthrough program for the 20 gigabyte, 500+ million polygon model of the Boeing 777 aircraft. Eric has also worked in computer graphics and CAD for Silicon Graphics, GRAFTEK, and the Jet Propulsion Laboratory. He holds eight patents, earned a BS and MS in mathematics and a PhD in applied mathematics from Rensselaer Polytechnic Institute, and is a certified performance technologist. Outside work, Eric has two teenage boys, the younger has autism. Eric works to develop autism insurance benefits, helps run the Microsoft autism alias, and serves on the board of the UW Autism Center and Families for Effective Autism Treatment of Washington. He enjoys time with his family, going to Mariners games, playing bridge, and driving his electric roadster. Although Eric shares I. M. Wright's passion for product, he tries to be a little more tolerant and open-minded.