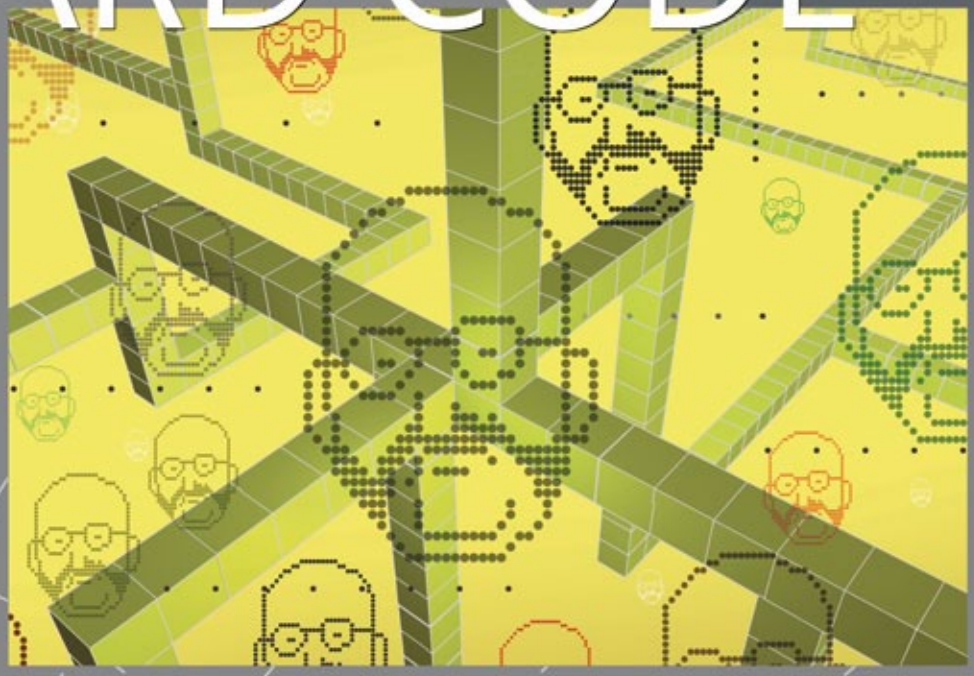
Microsoft

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

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.

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?

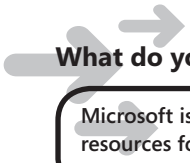
—Olof Hellman, Software Engineer

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



What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

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://www.microsoftpressstore.com/title/ 9780735661707

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:

http://www.microsoftpressstore.com/title/ 9780735661707

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.

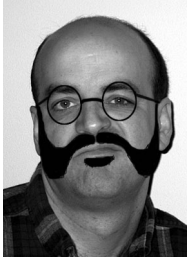
The customer wins

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.

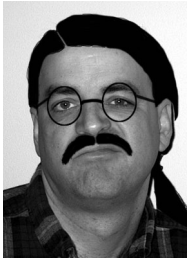
Eric Aside You can find my column on bug fields, priorities, and resolution values, called "Am I bugging you? Bug Reports," in Chapter 2.

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.