THE EXPERT'S VOICE®

Bill Blunden

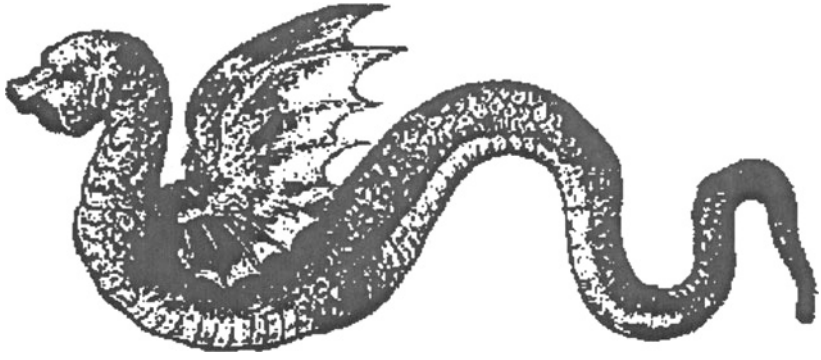
Software Exorcism

A Handbook for Debugging and Optimizing Legacy Code

Apress®

Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code

BILL BLUNDEN



Apress™

Software Exorcism: A Handbook for Debugging and Optimizing Legacy Code

Copyright © 2012 by Bill Blunden

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN 978-1-4302-5107-1

ISBN 978-1-4302-5108-8 (eBook)

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

President and Publisher: Paul Manning
Lead Editor: Dominic Shakeshaft
Technical Reviewer: Doug Holland
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Kari Brooks
Copy Editor: Ami Knox
Compositor: Kinetic Publishing Services, LLC
Indexer: Carol Burbo
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

This book is dedicated to bad kung fu movies.

Contents at a Glance

Contents

About the Author

Reverend Blunden has spent much of his life wandering through the subterranean catacombs of system software. At an early age, he discovered the DOS debug program, and he has been trying to peek behind the curtain ever since. Along the way, he received a degree in physics from Cornell University and a masters in operations research from Case Western Reserve University. Having narrowly escaped a lifetime of servitude to the Society of Actuaries, Reverend Bill fled westward. After ten years of searching in the desert, Reverend Bill was visited by an epiphany: a great smiling head named J. R. "Bob" Dobbs. Through Dobbs, Reverend Bill has learned the true nature of what makes software evil. He has come back to civilization to spread the word.

Acknowledgments

Writing a book is almost like being pregnant. For months you feel tired and sleep deprived. You have strange mood swings and food cravings. When you are finally done, you're exhausted and overjoyed that the tribulation is over. Yet people still continue to write, even after the first ordeal. I believe that the need to write must be an inveterate quality in some authors.

I would like to thank all the people at Apress who encouraged me and put up with all of my shenanigans. Specifically, I would like to thank Gary Cornell for giving me the opportunity to write for Apress. I would also like to thank Jim Sumser and Hollie Fischer for entertaining my lengthy diatribes and offering feedback on my dubious cogitations. Finally, I would like to thank Ami Knox, Kari Brooks, Beth Christmas, and Jessica Dolcourt for their assistance during the production process.

Praise Bob,
Rev. Bill Blunden
Church of the SubGenius

Introduction

IT HAS BEEN DOCUMENTED[1] that the maintenance phase of the typical software project's life cycle accounts for over 60 percent of the total cost incurred. For all the pomp and circumstance heaped upon the design phase, once the first release has been deployed, most of the resources will be spent on fixing bugs, adding new features, and fixing bugs resulting from those new features. The post-release development cycle of the average software application resembles that shown in Figure 1.

---

1. Don Coleman et al., "Using Metrics to Evaluate Software System Maintainability," *IEEE Computer,* August 1994, pp. 44–49

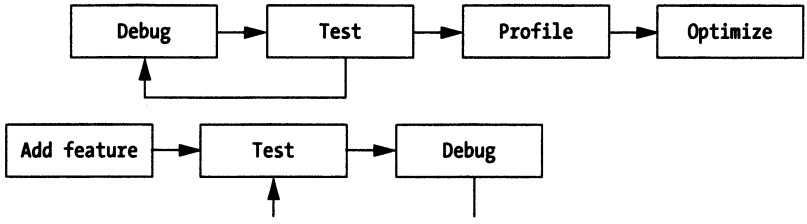


Figure 1. Post-release life cycle of the average software application

The post-release cycle can be initiated by the addition of a new feature or the submission of a bug report. In either case, unit and system tests are used to determine if the patched code is doing what it is supposed to. Once the paths of execution have been sufficiently stabilized, the application can be profiled to locate performance bottlenecks. A variety of optimization techniques can then be implemented to increase execution speed and decrease memory footprint. As customers demand new features and report new bugs, the cycle will repeat itself and the application will evolve.

Most books on software engineering focus intently on the phases of development leading up to the release of an application (e.g., requirements, analysis, use cases, implementation, and so on). Everyone knows that it's much easier, and more gratifying, to write a program from scratch. Software maintenance, on the other hand, is the ugly stepchild of computer science. Most professors would prefer not to mention it in public. This is because discussing the rigors of maintenance work might raise questions . . . dangerous questions. Why spend four years completing a degree in computer science when the fruits of your labor land you in an insanely frustrating maintenance role? If professors told their students the truth about software engineering as a profession, there would be a mass exodus out of computer science courses. A smaller department would translate into budget cuts and loss of prestige, so you'd better believe that the professors are more than willing to sidestep a few unpleasant truths to bolster the size of their classes.

This book is different. Rather than shield your eyes from the sordid realities of the software industry, I am going to dust off my old 8mm films and let you take a good look at the uncensored truth for yourself. You may want to keep a paper bag handy in case you get sick. This book examines the tools that engineers can use to facilitate the debugging and optimization of legacy software. In other words, this is a book on maintaining software that has been hacked to death by other people. Grab a bowl of popcorn and have a seat, but don't complain that I didn't warn you.

A Marine Corps Recon officer once told me that martial artists should not talk about techniques that are "combat effective" unless they have used those techniques under the threat of being killed. In this spirit, I feel that a book on software maintenance should not offer advice unless it has been used in the

field, by the author, under stressful circumstances. This book intentionally avoids ivory tower abstraction in favor of field-ready tactics. I stick to simple tools that have worked for me in the past. The average maintenance engineer does not have time to waste on cloud diagrams or fashionable development methodologies. Maintenance engineers have a job to do and an immediate need to get it done. This book presents tactics that may be used, in the trenches, with little or no preparation.

Historical Motivation

Back in the Iron Age (i.e., the late 1960s and 1970s), most software engineers did not have much computational real estate to play with. Room-sized computers like the CDC 6600 had only 65,000 60-bit words of memory (less than a megabyte). In this type of environment, every single bit counted, and engineers went to extraordinary lengths to save space. At the same time, processor speed was also a serious issue. Most people dropped off their programs with the system operator in the afternoon and came back the next day to pick up the results. Hence, engineers in the days of yore had to meticulously balance memory usage with the need to minimize the number of CPU cycles consumed.

This situation has changed. In 1998, I was working on a Windows NT workstation that had dual 450 MHz Pentium processors and 2GB of RAM. The CDC 6600 sold for $7,000,000 dollars. My NT workstation cost a little under $5000. Engineers today are not faced with the same pressures to squeeze every ounce of performance out of their programs (I can almost hear the CDC engineers grumbling, "Back in my day . . ."). To put it bluntly, we can be lazy if we want to and let the hardware pick up the slack. If a program doesn't run quickly enough, we can always throw more hardware at it, and with the emergence of cheap commodity parts, this is a realistic, short-term solution.

In the future, we are bound to hit a wall. The laws of physics require an electron to have a circuit path that is larger than three hydrogen atoms across. Once the path becomes smaller than this, the electron stops behaving like a particle and starts to behave like an escaped convict (i.e., quantum tunneling occurs). This means that the hardware industry will only be able to make computer processors shrink to a certain size. There will come a day when the processor manufacturers will no longer be able to have their cake and eat it too. At a certain point, increasing computing power will require processors to become larger in order to accommodate more transistors.

When this wall is hit, the responsibility for improvements will shift back onto the shoulders of software engineers. Better algorithms and better ways of implementing algorithms will need to be invented. The focus on optimization that occurred in the 1960s will reemerge as the next generation of pioneers pushes the envelope for application performance.

NOTE *When exactly will we hit the wall? I'm sure a lot of chip vendors don't like to think about it. In 1989 the Intel 80486 was released with a design rule of about a micrometer, which is a millionth of a meter. The anthrax bacterium is roughly 1 to 6 micrometers in length. A human hair is about 100 micrometers in diameter. According to Gordon Moore's observation, known as* Moore's Law, *the number of transistors in a given area doubles every 18 months. In other words, the design rule of a transistor should be cut in half every 3 years. If we take 1989 as a starting point, where the design rule of a transistor was 1 micrometer, then you should be able to plug through the math and see that the show will be over by 2022. Even if CPU vendors drag things out, I doubt if things will continue to shrink after 2100.* Also sprach Zarathustra.

During the Iron Age, debugging often entailed reading hex dumps and reconstructing stack frames. The very act of looking for a bug was painfully time consuming. As a result, engineers developed precise strategies for preventing bugs and fixing them quickly when they occurred. Today, most engineers simply place a breakpoint somewhere in their code and start stepping through their program with a GUI debugger. GUI debuggers and shotgun surgery have replaced the detective skills that were so vital to the older generation of engineers.

I'm not saying that GUI debuggers are bad; in fact, I'm a huge fan. I'm just saying that there are times when a GUI debugger by itself won't illuminate the source of a problem. In these situations, what's called for is a set of diagnostic skills that must be honed through disciplined study and practice. These skills, in addition to the GUI debugger, can be used to address tough problems.

Using tools effectively is not a bad thing, unless you become completely dependent upon them. This reminds me of a story that Isaac Asimov wrote called "The Feeling of Power." In this story, future generations become so dependent upon computers that they forget how to perform basic arithmetic. At the end the story, one of the main characters performs multiplication in his head:

> *Nine times seven, thought Shuman with deep satisfaction, is sixty-three, and I don't need a computer to tell me so. The computer is in my own head. And it was amazing the feeling of power that gave him.*

As time progresses, software applications will become larger and larger. The first version of PC DOS was roughly 4000 lines of assembly code.[2] That

2. Andrew Tanenbaum, *Modern Operating Systems*, Second Edition (Prentice Hall, 2001. ISBN: 0-13-031358-0)

was back in 1981. Fast forward to the mid 1990s. Windows NT 4.0 was over 16 million lines of code.[3] In the coming century, we will probably see software applications that top a billion lines of code. Although the current generation of debugging tools is impressive, the casual approach assumed by contemporary developers will not suffice as the complexity curve ramps up towards the stratosphere.

Even if the optimization and debugging skills of the average software engineer have atrophied, in the advent of superior hardware and tools, engineers worth their salt will still take the time to master these forgotten arts. The investment of time and energy will pay themselves off by rewarding the investor with skills that will make them a better engineer. There is nothing worse than being woken up at 3:00 in the morning by an angry client. With this book, you can protect yourself from this kind of interruption and get back to sleeping nights.

Audience

According to the U.S. Bureau of Labor Statistics, there were over a million software engineers employed nationwide in 2001. Easily half of these engineers performed maintenance work of some form or another. Hence, this book targets a sizeable cross-section of software professionals.

Maintenance programming is not a glamorous position. It's more like working in a steel mill: tedious and fraught with danger. The high-paid consultants, whom management recruited specifically to perform the first cut, run from maintenance work like the plague. Why? Well, consulting architects avoid maintenance because it sucks. It's tedious, frustrating, and completely uninspired. That's why they gave it to you, the new guy.

The average maintenance engineer will usually be given a few thousand lines of code, a brief explanation, and then directed to make improvements. They rarely have had the benefit of being a part of the original development team and are often confronted with even stricter deadlines. This book is dedicated to such maintenance programmers, who slog away in dimly lit cubes, drink day-old coffee, and wonder silently to themselves, "How on earth did I ever get into this mess?"

Maintenance engineers of the world: I feel your pain.

Organization

Software, as it is developed today, tends to follow a series of basic dance steps that include construction, testing, debugging, and fine-tuning. A number of models describe the general process. For instance, the venerable waterfall approach (see Figure 2) is one of the oldest models for describing how these development dance steps are ordered.

---

3. Don Clark, "Windows NT Is Back," *The Wall Street Journal*, July 29, 1996

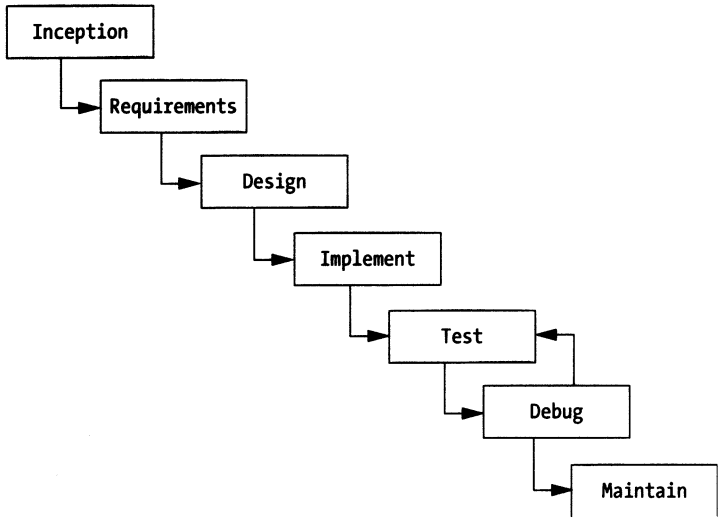


Figure 2. The waterfall development model

Many engineers scoff at the waterfall model, and with good reason. In today's environment, where businesses age in Silicon Valley years, there are really only two steps in the process, implementation and maintenance, and the people who perform the implementation have all made a run for the border.

This book consists of seven chapters. I spend the very first chapter trying to spell out a few tricks that can possibly save you from reading the next three chapters—which is to say that I offer a few pointers on how to write code that is bug resistant. If you are not in the enviable position of writing the first cut of a software program, then you can ignore the first chapter. Chapters 2 through 4 are dedicated to the high art of debugging. I not only look at strategies for debugging code, but I also examine how debuggers function internally.

Besides debugging, most maintenance engineers spend their workday fine-tuning programs to improve their performance. Chapters 5 and 6 are devoted to explaining ways to make more efficient use of a computer's resources. I end the book with an abbreviated final chapter that offers a few words of hard-won advice.

Following is a more detailed rundown of what each chapter covers.

Chapter 1: Preventative Medicine

The original authors of a program are almost always responsible for most of the bugs. Engineers, in the privileged position of building a piece of software from scratch, have the unique opportunity to institute conventions that minimize the number of bugs that they embed in their creation. This chapter is

devoted to examining a set of techniques that can be used to construct software that is easy to modify, and hence easy to debug.

Chapter 2: Debugging Tactics

This chapter presents a step-by-step approach for locating and eliminating software bugs. I begin by discussing how to verify that the issue you are dealing with is actually a bug. Next, I examine the scientific method and explain how it can be applied to deal with bugs. To persist in your work and provide a trail guide for the engineers who follow after you, this chapter looks at the methods that can be used to track maintenance work.

Chapter 3: Understand the Problem

In Chapter 2, I mention that understanding the problem, and program, that you are working with is a prerequisite. But how do you "understand the problem" when you are confronted with 50,000 lines of cryptic Kernighan and Ritchie C? Aha! That is the $64,000 question. In this chapter, I provide a number of battle-tested methods that you can use to address this question. At the end of the day, there is no replacement for hard work. However, there are steps you can take to make your job easier. I can only say, in my own defense, that it is easier to write about this topic than it is to actually do it.

Chapter 4: Debugger Internals

Understanding how a debugger works, underneath the hood, is not required to use a debugger successfully. However, there may be a few curious readers who have a nagging desire to know how debuggers operate. This chapter is devoted to explaining the essentials of how debuggers do what they do. I begin with basic features, like breakpoints and single stepping, and then gradually move into more advanced functionality. I conclude this chapter with a discussion of techniques that can be used to protect your program from being reverse engineered.

Chapter 5: Optimization: Memory Footprint

Computers have two resources: memory and CPU cycles. Successful optimization depends upon being able to both minimize the memory footprint of an application and make efficient use of processor cycles. It is a very delicate balancing act. This chapter looks at the first half of this puzzle by offering techniques for reducing the amount of memory that an application uses. All of the standard memory components of a program are examined, including the code segment, data segment, stack, and heap.

Chapter 6: Optimization: CPU Cycles

This chapter is the second half of the discussion that began in Chapter 5. Specifically, Chapter 6 presents a number of ways in which programs waste processor cycles, and then offers solutions in each instance. I begin by analyzing elementary program control constructs, like loops and branching statements, and then move on to more advanced topics, like exception handling and memory management.

Chapter 7: Final Words of Advice

Some things your professors in school will not warn you about, primarily because they can't: they have spent most of their professional lives in academia. There have been those, however, who left the cocoon-like safety of the university to brave the elements in the software industry. In this chapter, I tell you a few things that I, and others like me, have learned.

Typographical Conventions

In this book, the courier font is used to indicate that text is one of the following:

- Console output
- A filename or file extension type
- Source code
- Numeric values

Hexadecimal values are indicated, in this book, by prefixing them with a `0x`. For example, the integer value `5689` will, in hexadecimal, look like `0x1639`.

Words will appear in the *italic* font, in this book, for two reasons:

- When defining a new term
- For emphasis

Prerequisites

The examples in this book are primarily implemented in a combination of ANSI C, C++, and ×86 assembly language. I rely on C/C++ and assembler not only to appeal to the largest number of programmers, but also to provide insight. To see why a certain strategy is effective, there will be times when you

will need to see what the compiler is doing behind the scenes. The best way to do this is to take a look at the assembly code that the compiler generates. Having said that, my use of assembly code is fairly pedestrian, and I think that most engineers who have spent any time behind the wheel will be comfortable reading through my code.

Initially, I thought about using Java to implement examples. However, I have found that the flexibility of C and C++ provides greater opportunity for misuse and mistakes. What did Spider-Man say? With great power comes great responsibility? C and C++ are powerful languages. Hence, with C/C++ it's easier to make a mess. Memory leaks and dangling pointers have beset C programmers for decades, not to mention preprocessor pyrotechnics and arbitrary levels of indirection. In short, C and C++ provide fertile territory for discussing debugging and optimization.

I have often listened to debates concerning the relative merits of C++ and Java. As far as I am concerned, these two languages are different tools used for different jobs. It is like asking, "Which is better, a pencil or a pen?" Both languages are object oriented, but the primary difference between them lies in their orientation. Specifically, Java is an *application language* and C++ is a *system language.*

Java programs are compiled to run on a virtual machine. "Write once, run anywhere" is the fundamental benefit of implementing a project with Java. This feature has made the language extremely popular with software companies, like IBM, that strive for cross-platform portability. The downside to this is that you cannot directly interact with native hardware. By striving for portability, Java has insulated itself from the host machine.

Likewise, building system software requires that you have the ability to do things like insert inline assembly code, explicitly manipulate memory, and generate a native executable. It just so happens that C++ provides these features. It is no surprise, then, that almost every operating system currently in production is written in a combination of C and C++. When it comes to handling interrupts and communicating with hardware, there is always some assembly code involved—it's unavoidable. C and C++ allow high-level constructs and assembly code to intermingle so that assembly-level routines can be hidden away in the bowels of the system.

CHAPTER 1

Preventative Medicine

Quite frankly, I'd rather weed out the people who don't start being careful early rather than late. That sounds callous, and by God, it _is_ callous. But it's not the kind of "if you can't stand the heat, get out of the kitchen" kind of remark that some people take it for. No, it's something much more deeper: I'd rather not work with people who aren't careful. It's Darwinism in software development.

—Linus Torvalds on kernel debuggers, Linux Kernel Mailing List

The role of the maintenance engineer is to exorcise the evil spirits that dwell in legacy software. Day and night, maintenance engineers are beset upon, and spited, as they forge a lonely path through the corporate landscape. Every day, they face the challenge of

- Repairing bugs
- Improving performance

The first four chapters of this book are devoted to looking at the first task. The final two chapters of this book deal with the second task.

An 1896 electrical handbook entitled Hawkin's *New Catechism of Electricity* states that "The term 'bug' is used, to a limited extent, to designate any fault or trouble in the connections or working of electric apparatus."[1] This term evolved, with a little help from Rear Admiral Grace Murray Hopper (the inventor of COBOL), to denote a malfunction in a computer program.

Dealing with bugs is one of the most odious tasks that a programmer can perform. Given this fact, the best time to deal with a software bug is during the implementation phase. In other words, fix the problems before they become a part of the release build. This saves maintenance engineers from having to deal with them later on, after the bugs have had the chance to burrow deeply and hide. Many professionals, including Linus Torvalds, agree with this train of thought.

Sun Tzu once said, *"You should do something large while it is still small."* In other words, have the foresight to implement corrections and isolate problems early. This way you won't have to worry about getting a phone call at 3:00 in the morning from an angry system administrator.

In this chapter, I will examine preventative measures that can be used to make source code less susceptible to bugs. If you are an engineer in the process of building a brand new piece of software, or adding a new feature, you should read this chapter carefully. On the other hand, if you have been handed an application that you did *not* write (and this is a very common occurrence), then I would recommend that you skip this chapter initially, and go to the next chapter. Once you have dealt with the offending bug, you can come back and use the techniques in this chapter to help you clean up your patch.

1.1 Core Problems

You may be smirking to yourself right about now: "Catch bugs before they occur? Ha, if it were only that simple!" The real world is rife with external pressures that confound this type of measure-twice-cut-once approach to software development. In this section, I will examine real-world problems that can lead to the construction of buggy code, and offer possible remedies when I can.

---

1. Eric S. Raymond, ed., *The New Hacker's Dictionary, Third Edition* (MIT Press, 1996. ISBN: 0-262-68092-0)

1.1.1 Time to Market Pressure

The problem with using disciplined engineering practices to catch program bugs is that they require extra effort, and extra effort costs time. Time is a valuable commodity in the post dot-com era. Software companies are under intense pressure, from their backers, to beat their competitors to the punch. They aren't given the time to invest in a solid development scheme. Heck, even if they did have the time to utilize a sane development methodology, many companies wouldn't know how because these same companies usually try to save money by hiring inexperienced developers. All that matters is getting a product out the door as soon as possible, preferably under budget. Yes, you're right, it is stupid. Welcome to the business world.

Haste Makes Waste

While the time-driven approach may garner results from a marketing perspective, where perception is everything, over the long run it is a losing strategy. *Haste makes waste,* to the extent that it has been designated as a design antipattern.[2] When software engineers rush to get things done, they work late, they take shortcuts, and this results in bugs. It's as simple as that.

Who cares if a company is the first to release a product, especially if the product being sold is so bug-ridden and clunky that it alienates users? Being first doesn't matter if your execution sucks. It doesn't matter how much buzz the marketing people can generate.

Let's look at a case study. When IBM released OS/2 version 2.00 in 1992, they touted the fact that it was the first true 32-bit operating system for the PC. OS/2 supported legacy DOS applications via a 16-bit subsystem (i.e., a DOS Virtual Machine) and Windows applications on IBM's licensed version of Windows 3.0 (i.e., Win-OS/2). The problem was that the compatibility code was so poor that it forced DOS and Windows applications to slow to a crawl. Not to mention the perplexing peripheral interface, which was an absolute disaster. Adding a new printer on OS/2 was so involved that people literally wrote entire magazine articles on how to do it.

Strictly speaking, OS/2 came to market way before Windows NT 3.1, which was released in 1993. IBM may have beat Dave Cutler to market, but being early didn't help much. OS/2 performed so poorly that customers who could afford 16 megabytes of RAM in 1993 opted for NT. At the time of this

---

2. William J. Brown et al., *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* (John Wiley & Sons, 1998. ISBN: 0-471-19713-0)

book's publishing, IBM is still selling OS/2 Warp 4. Can you believe it? I know an IBM consultant who looked me straight in the face (without smiling) and said, "Inside IBM, OS/2 is still seen as a viable desktop operating system."

Logic Is Not a Factor

After four years of truth tables, and algorithms, and mathematics, most computer science majors expect the world to be run by people who make rational decisions. BZZZZT, sorry, wrong answer. While the argument for using a disciplined development process makes sense, not everything in the business world makes sense. Business decisions often get made that have very little to do with logic. For example, did you know that IBM still sells a version of DOS called PC DOS 2000?

Politics, charisma, and corporate culture can be just as influential as logic. To make matters worse, the executive officers who wield these intangible powers of persuasion tend to have considerably less technical expertise than the engineers. If the slick marketing executive can somehow convince the CEO that they have to get project XYZ to the public before anyone else, then you can expect to be asked to sacrifice everything in the name of expediency. Such is life.

Ultimately, the decision making process in a corporation is run by the golden rule: the person who has the gold makes the rules. Put another way, the person who controls the resources ultimately makes the final decision. Money talks, and logic takes a backseat. You can sulk all you want, but this is how the world runs. Recognizing this will help you understand why things like ridiculous project schedules occur.

Being First Can Backfire

Sometimes being first is actually hazardous, particularly if you get the attention of a larger competitor. It is the pioneers in an industry that have to jump all of the initial hurdles to validate a new product and establish an audience. Large multinational corporations often wait back in the wings, letting the venture startup do the footwork to blaze a new trail. Once the new market has proven to be profitable, the multinationals move in with their salesmen and economies of scale. If the pioneers are lucky, a competing multinational might buy them out.

Let's look back at history again. In December 1994, Netscape released the first version of its Navigator Web browser. This caught the attention of Bill Gates, who not only realized that Netscape was on to something, but that it also threatened his business. Subsequently, Gates unleashed his army of developers and in August 1995 Internet Explorer (IE) version 1.0 was released as a part of the Windows 95 PLUS pack. Not only that, but the price for IE was right. As everyone knows, it is not wise to stand in the way of an oncoming

18-wheel Mack truck. Likewise, it is not very bright to get in the way of Bill Gates. In 1998, after several years of assault, America Online (AOL) acquired Netscape in a transaction worth $4.2 billion.[3] Like I said: if the pioneers are lucky . . .

The moral of this story is not that you shouldn't be first because a larger competitor will crush you. Rather, I believe that if you are going to sell a software application, then you should make sure that it has every advantage possible so that it can stand apart from similar products. This includes taking the time and investing the resources necessary to ensure application stability.

Increased Competition

How did things get so competitive? Why the rush to market? Whatever happened to Ward Cleaver and his nine-to-five job? Well, I don't have a definite answer, but I can speculate. I am of the opinion that the emergence of pervasive networking has allowed consumers to become pickier. This may sound simpleminded, but I think that the aggregate impact of well-informed consumers is more significant that you might suspect. What used to be a seller's market is now a buyer's market.

For example, back in 1970, most consumers trying to buy a television were stuck with the vendors in their immediate vicinity. Finding different purveyors and comparing prices meant digging around for information that was difficult to locate. Today, if you don't like the price that the dealer down the street is selling at, you can crank up a Web browser and check out price information on eBay. Data that was once very time-consuming to obtain is now very easy to acquire.

1.1.2 Fluid Specifications

Sometimes people don't really know what they want, even after they have told you. In pathological cases, the sponsor of a software project may keep adding new requirements while the engineers are in the process of implementing that project. This is known as *feature creep.* It is an insidious project killer.

Feature Creep Causes Premature Code Decay

Most software programs can stand only so much modification before they need to be completely redesigned. Another way of saying this is that change

---

3. Rebecca Sykes, "AOL buys Netscape, joins Sun in Java Deal," IDG News Service, November 24, 1998

can damage the integrity of a program's design. Modified code becomes brittle. By initiating this gradual decay early, during the implementation phase, feature creep makes life more difficult for the maintenance engineer. It's like moving into a new house, only to find that squatters have already lived there and made a few renovations of their own.

There Is Always Something New

The early adoption of new technology is one of the most common causes of feature creep. In the software industry, in general, some new development framework is always waiting around the next corner. It is the nature of the beast. Technophiles, or people who are stalling for time, will often "discover" a new gizmo and then demand that new requirements be added for it (with the justification that it allows for future flexibility). In terminal cases, a project will *never* be deployed because each time it nears code cutoff, a new technology is discovered.

1.1.3 Trade Features for Time

How exactly should an engineer cope with unreasonable deadlines and feature creep? The best way to deal with deadlines is to quantify them in terms of application features. The longer you have to work on a project, the more features you will be able to implement. Likewise, the less time you have to work on a project, the fewer features you will be able to implement. If a project sponsor wants to stick you with a tight schedule, then you should accept the schedule only after doing an analysis to determine which features you can successfully implement in that period of time.

The alternative is to spend the next six months pulling 15-hour days, trying to meet a deadline that may be completely unrealistic. This type of deadline basically forces you to exchange your life for a paycheck, which is a bad deal by any standard. No one lies on their deathbed wishing they had spent more time at the office. If you are a software engineer spending 15 hours a day at work, you need to get a life. Sitting in front of a computer for weeks on end may seem heroic, at first glance, but it is a losing strategy over the long run. Have you ever looked at software that was written by someone subsisting on 3 hours of sleep?

If project sponsors approach you with a request to add a new feature while you are knee-deep in the implementation phase, you should make them aware that this new feature will require the timetable to be adjusted to accommodate the new feature. In other words, tell sponsors that they will not get the new feature for free; it will cost them time. Time translates into money, and money is a language that most business people understand. This is the key to dealing with executives: find a way to speak to them in their native language (i.e., $$).

1.1.4 Get It in Writing

There is a sinister little game that Marine Corps drill instructors sometimes play on candidates in basic training. One day, while a candidate is nervously standing at attention, a drill instructor walks up and says something like this:

"Private Blunden, the other drill instructors tell me that you are a pretty smart guy. Well, private, I have decided that I would like to see this for myself. Private Blunden, I'd like you to demonstrate your renowned brilliance and raise your hand for me."

This is a setup. When the candidate raises one of their hands, the drill instructor will yell at the top of their lungs:

"I meant the other hand, Blunden, you stupid SOB! You raised the wrong *@#$% hand!"

This sadistic trick teaches an important message. Always ask for clarification if you feel like the directions that you have been given are ambiguous. Otherwise, someone can accuse you of doing the wrong thing.

If someone really wants to stick it to you, they will intentionally give you ambiguous orders so that they have an excuse to attack you when you don't do things their way. In other words, both answers are wrong, and they're just looking for an excuse to abuse you. You could breathe air and they'd start criticizing you for breathing out of your nose instead of your mouth.

Nail Down Requirements

You should never make an assumption about a software requirement that you think is ambiguous. Always encourage customers to tell you exactly what it is they want, even if you have to drag it out of them. Better that you annoy them up front than have to spend weeks reimplementing a core product feature.

This is your responsibility as an engineer. Do not sit passively hoping that someone will magically appear at your cube and tell you everything that you need to know. Don't fall into this trap—and it is a trap. You need to be aggressive. If you aren't clear on an issue, relentlessly hunt down the truth until you find it. No one else is going to do if for you, so if you just sit in your cube, scratching your head, you are wasting valuable time.

To be honest, a lot of people have a problem with this, including me. I prefer to be given a well-defined task and then sit down and work on it. I find it aggravating when I have to run around, bouncing from one person to the next, to try and get a straight answer from someone. To me, this seems like a huge waste of time and money. Like I said, the business world does not always make sense.

Get a Signature

Once you have extracted a set of solid requirements, the next step would be to record them as a document and have the project sponsor sign them. This document now constitutes a contract that binds the involved parties. The benefit of such a contract is that it establishes a common set of expectations and decreases the likelihood of a misunderstanding.

But why get a signature? It's not like anyone is getting married? Or is it . . .

Foil Plausible Deniability

During your career you may meet people, particularly higher-ups in the food chain, who do not like to put anything in writing. There is a reason for this preference, a reason that may not be immediately recognizable to more decent and honest members of the audience. The truth is that anything in writing creates a paper trail. Well-maintained paper trails can be used to track down the people responsible for making certain decisions. Responsibility can translate into blame, and blame translates into unemployment. Understand?

> *To err is human. To cover it up is weasel.*
>
> —Scott Adams, *Dilbert and the Way of the Weasel*

The business world has its share of weasels. If the project that you are working on heads south, less scrupulous people may be tempted to switch sides and ridicule a project that they once championed. This reminds me of people who always cheer for the winning team in a play-off series. In a less drastic scenario, a manager may simply deny any responsibility for failure and lay the blame on someone else's shoulders. "It's not my fault, Mr. Gerstner, everyone knows that those guys over in manufacturing couldn't keep to the specifications." Don't think that this doesn't happen. A good paper trail will safeguard you from weasels like this.

People with a propensity to "abandon ship" prefer verbal specifications to written specifications, particularly when there are no witnesses to verify the verbal specification. If push comes to shove, they can always make it their word against your word. This is why you want to bind them with a signature, preferably during a meeting, where plenty of other people are around, in a room with CCTV cameras.

If you cannot get a signature, then at least make sure to schedule a meeting and invite everyone who has a stake in the project. During the meeting you can nail down requirements without having to demand that the project

sponsor sign anything in blood. However, make sure to designate someone to transcribe the meeting and then send that transcription, via e-mail, to all of the participants. If people are not adverse, you can opt to tape-record the meeting so that no one is stuck as a transcriber.

In the worst-case scenario, you may be dealing with a *ghost.* In other words, the people dictating requirements may be able to elude making public statements. Or, if they do, they will never say anything incriminating. Ghosts tend to have political savvy and years of experience playing the game. This makes them dangerous. Ghosts prefer to corner you in the hallway and dictate their requirements during an impromptu conversation. This backroom approach to deal making is favored by people who don't like being quoted. Can you spell "plausible deniability"?

Use E-Mail

If you cannot schedule a formal meeting, then follow up the conversations with e-mail. Make sure to send copies of the e-mail to others involved with the project. You want the ghost to know that the conversation is now public knowledge. This will prevent sudden bouts of amnesia. *The basic idea behind all of this is that you need to create some sort of artifact.* If the waters become stormy, you will need evidence to protect yourself. Your credibility depends upon it. Maintain a paper trail and then make copies.

Buy a Home Safe

Once you have generated a paper trail, take it home at the end of the day and store it in a safe place. I prefer a 700-pound TLTR-30x6 vault, which can withstand tools and a blowtorch for 30 minutes on all six sides.[4]

This may sound a little paranoid, but it's not. Trust me. Shredding documents to destroy the paper trail is a time-honored tradition in the business world (uh, ahem . . . Enron, uh, ahem . . . Arthur Andersen). Your desk, chair, computer, and office supplies at work aren't really yours; they are tools for you to do your job and are the property of your employer. There is nothing that prevents a manager from walking up to your desk and taking every last bit of evidence to the garbage bin.

---

4. Gardall Safe Corporation, P.O. Box 240, Syracuse, NY 13206-0240. Phone: 1-800-722-7233