Hacknot

Contents

Foreword vi

Peopleware 1

The A to Z of Programmer Predilections 2
The Hazards of Being Quality Guy 16
A Dozen Ways to Sustain Irrational Technology Decisions 19
My Kingdom for a Door 27
Interview with the Sociopath 35
The Art of Flame War 42
Testers: Are They Vegetable or Mineral? 50
Corporate Pimps: Dealing With Technical Recruiters 56
Developers are from Mars, Programmers are from Venus 64

Management 71

To The Management 72
Great Mistakes in Technical Leadership 75
The Architecture Group 87
The Mismeasure of Man 90
Meeting Driven Development 102

Extreme Programming and Agile Methods 109

Extreme Deprogramming 110
New Methodologies or New Age Methodologies? 116
Rhetorical AntiPatterns in XP 121
The Deflowering of a Pair Programming Virgin 129
XP and ESP: The Truth is Out There! 134
Thought Leaders and Thought Followers 139

Requirements 141

Dude, Where's my Spacecraft? 142
User is a Four Letter Word 145

Design 149

The Folly of Emergent Design 150
The Top Ten Elements of Good Software Design 157

Documentation 162

Oral Documentation: Not Worth the Paper it's Written On 163
FUDD: Fear, Uncertainty, Doubt and Design Documentation 168

Programming 174

Get Your Filthy Tags Out of My Javadoc, Eugene 175
Naming Classes: Do it Once and Do it Right 178
In Praise of Code Reviews 183

User Interfaces 189

Web Accessibility for the Apathetic 190
SWT: So What? 196

Debugging and Maintenance 206

Debugging 101 207
Spare a Thought for the Next Guy 221
Six Legacy Code AntiPatterns 223

Skepticism 227

The Skeptical Software Development Manifesto 228
Basic Critical Thinking for Software Developers 232
Anecdotal Evidence and Other Fairy Tales 238
Function Points: Numerology for Software Developers 244
Programming and the Scientific Method 252
From Tulip Mania to Dot Com Mania 260

The Industry 263

The Crooked Timber of Software Development 264
From James Dean to J2EE: The Genesis of Cool 270
IEEE Software Endorses Plagiarism 273
Early Adopters or Trend Surfers? 278
Reuse is Dead. Long Live Reuse. 280
All Aboard the Gravy Train 282

Foreword

The Hacknot Web Site

Hacknot began life in 2001 as an internal mailing list at the multinational telecommunications company I was then working for. As part of the activities of the local Software Engineering Process Group, I was looking for a way to promote discussion amongst staff about software engineering related issues, and hopefully encourage people to learn about the methods and techniques that could be used to improve the quality of their work. A creative colleague came up with the name "Hacknot" for the mailing list … a pun on the geek web site "slashdot."

A few years later, when I left the company, I restarted Hacknot as an externally hosted mailing list, with many of the same members as in its last incarnation. In 2003, I was looking for a coding exercise in J2EE, the main technologies of which had passed me by while I was busy working in other areas. Growing tired of building play-applications like bug trackers and online store simulations, I decided to create a web version of the Hacknot mailing list. I figured it would give me a "real world" context in which to learn about J2EE, and also a project that I could pursue without the interference of the usually inept management that so plagued the development efforts of my working life.

So in 2003 the Hacknot web site was born. In Australia, domain name registration rules restrict ownership of ".com" domains to commercial enterprises, so I chose the next best top-level domain, which was ".info".

Initially, I imagined that the web site would host works by a variety of authors, myself included. But when it came time to put pen to paper, almost all of those who had previously expressed interest in participating suddenly backed off, leaving me to write all the content myself.

Many of the essays on Hacknot take a stab at some sacred cow of the software development field, such as Extreme Programming, Open Source and Function Point Analysis. These subjects tend to attract fanatical adherents who don't take kindly to someone criticizing what for them has become an object of veneration. The vitriol of some of the e-mail I receive is testament to the fact that some people need to get out more and get a sense of perspective. It is partially because of the controversial nature of these topics that I have always written behind a pseudonym; either "Ed", "Mr. Ed" or "Ed Johnson". I also favor anonymity because it makes a nice change from the relentless self-promotion engaged in by so many members of the IT community.

The Hacknot Book

This book contains 46 essays originally published on the Hacknot web site between 2003 and 2006. The version of each essay appearing in the book is substantially the same as the online version, with some minor revisions and editing.

You can freely download copies of this book in PDF format with page sizes of 6” x 9” or A4, by visiting `http://www.hacknot.info`. There you will also find instructions on how you can obtain a paper-back copy, for the price of the binding and postage.

Please send any comments or corrections to `editor@hacknot.info`.

Ed Johnson
Sydney, Australia
December 2006

Peopleware

The A to Z of Programmer Predilections*

There is a realization that comes with the accrual of software development experience across a reasonable number of organizations, and it is this:

Though the names change, the problems remain the same.

Traveling from project to project, from one organization to another, across disparate geographies, domains and technologies, I am repeatedly struck more by the similarities between the projects I work on than their differences. Scenes from one job seem to replay in the next one, only with a different set of actors.

You might finish a gig in which you've seen a project flop due to inadequate consultation with end users, only to find your next project heading down the same path for exactly the same reason. And it generally doesn't matter how much you jump up and down and try and warn your new project team that you've seen the disastrous results of similar actions in the past. They will ignore you, insisting that their situation is somehow different. You will stand back and watch in horror as the whole scenario plays out as you knew it would, all the while unable to do anything more to prevent it. The IT contractor's career can be like some cruel matinee of "Groundhog Day" – without the moral resolution at the end.

But this technological déjà vu is not limited to technical scenarios - it extends to people. I find myself working with the same programmers over and over again. Their names and faces change, but their personalities and predilections are immediately recognizable. I find myself playing mental games of "Snap" with my fellow developers. "Bob over there is just like Ian from Acme. James is this workplace's equivalent of Charles from that financial services gig I had last year" – and so on.

Sometimes I fancy that I have met them all. There will be no new developers for me to work with in future – only the reanimated ghosts of projects past. The same quirks and foibles that I've endured in the past will haunt me the rest of my days.

I've listed below the cast of characters that have been following me around for some years now. Coincidentally, there are exactly twenty six of them, one for each letter of the alphabet. Perhaps you've encountered some of them yourself. Perhaps you're one of them. If so – please go away and find someone else to bug.

Arrogant Arthur

The three hardest words in any techie's vocabulary are "I don't know". Arthur never has to struggle with them, for he knows everything. Any technology you might name - he's an expert. Any problem you might have – he's solved it before. No matter what challenge he's assigned – he's sure it will be easy. Whenever Arthur appears to have made a mistake, closer investigation will reveal that the fault in fact lies with someone or something else. Arthur is a pretty handy conversationalist. Whenever you're having a technical discussion with someone and he is within earshot, Arthur will generally join in and quickly dominate the discussion with his displays of erudition. Uncertainty and self-doubt are states of mind that Arthur is entirely unfamiliar with. Arthur has a tendency to make big generalizations and sweeping statements, as if to imply that he has the certainty that only comes from vast experience.

Belligerent Brian

Nobody in the office is particularly fond of Brian. Sure, he's a smart guy and seems to be technically well informed, but he has such a strident and aggressive manner that it's difficult to talk with him for any length of time without feeling that you are under attack. Brian likes it that way and his hostile manner is entirely intentional. You see, Brian is a go-getter. Highly ambitious and energetic, he is determined to advance up the corporate ladder, no matter who he has to step on in the process. Whenever any action is undertaken or decision made, there is always a part of him thinking "How will this make me look to my manager?" It's not surprising then that not all of Brian's decisions are good ones. He has been known to select cutting edge technologies simply for their buzzword compliance, betting that cool acronyms and shiny new methodologies will make him appear progressive and forward-looking. Although he regularly makes mistakes, Brian never admits to any of them, and generally blames third parties, vendors and colleagues for errors that are actually his own.

C++ Colin

Colin is the local language bigot, whose language of preference is C++. He began programming in C, moved on to C++ when commercial forces threw the OO paradigm at him, and has been working in C++ ever since. Colin has watched the ascent of Java with a mixture of disdain and veiled jealousy. Initially, it was easy to defend C++ against criticisms from the

Java camp, by pointing to C++'s superior performance. But with the growing speed of JVMs, this advantage has been lost. Now, most of the advantages that Colin claims for C++ are the same language features that Java enthusiasts see as disadvantages. Java developers (or, "Java weenies" as Colin is fond of calling them) point to automatic memory reclamation as an eliminator of a whole category of bugs that C++ developers must still contend with. Colin sees garbage collection as disempowering the programmer, referring to the random intrusion of garbage collection cycles as payback for those too lazy to free memory themselves. Java weenies consider the absence of multiple inheritance in Java an advantage because it avoids any confusion over the rules used to resolve inheritance of conflicting features; Colin sees it as an unforgivable limitation to effective and accurate domain modeling. Java weenies consider C++'s operator overloading to be an archaic syntax shortcut, rife with potential for error; Colin sees it as a concise and natural way to capture operations upon objects. Colin displays a certain bitterness, resulting from the dwindling variety of work available to him within the language domain he is comfortable with.

Distracted Daniel

Daniel's mind is only ever half on the job, or to put it another way, he doesn't have his head in the game. Daniel lives a very full life – indeed, so full that his private life overflows copiously into his professional one. He has several hobbies that he is passionate about, and he is always ready to regale a colleague with tales of his weekend exploits in one of them. It looks as if his job is just a way of funding his many (often expensive) hobbies. His work is strictly a nine to five endeavor, and it would be very rare to find him reading around a particular work-related topic in his own time, or putting in an extraordinary effort to meet a deadline or project milestone. He is constantly taking off at lunch times to take care of one task or another, and does not seem to be particularly productive even when he is in the office. Daniel refers to this as "leading a balanced life". He may be right.

Essential Eric

Eric knows that knowledge is power. Partly by happenstance but mostly by design, Eric has become irreplaceable to his employer. There just seems to be a vast amount of technical and procedural arcana that only Eric knows. If he should ever leave, the company would be in a mess, as he would take so much critical information with him. This gives him a good deal of bargaining power with management, and good job security. A few of the company's managers have recognized the unhealthy dependence that exists upon him, and have attempted to document some of the valuable knowledge about certain pieces of software central to the business, but Eric always finds a way to get out of it. There always seems to be something more pressing for him to do, and if he is forced to put pen to paper, what results tends to be incoherent nonsense. It seems that he just can't write things down - or rather, that he chooses to be so poor at it that no one even bothers to ask him to document things any more. Eric is not keen to help others in those domains that he is master of, as he doesn't want to dilute the power of his monopoly.

Feature Creep Frank

Most of the trouble that Frank has got himself into over the years has been heralded by the phrase "Wouldn't if be cool if ... ". No matter how feature-laden his current project may be, Frank can always think of one more bell or whistle to tack onto it that will make it so much cooler. Having decided that a particular feature is critical to user acceptance of the application, it is a very difficult task to stop him adding it in. He has been known to work nights and weekends just to get his favorite feature incorporated into the code base – whether he has got permission to do so or not. Part of Frank's cavalier attitude to these "enhancements" comes from his unwillingness to consider the long term consequences of each addition. He tends to think of the work being over once the feature has been coded, but he fails to consider that this feature must now be tested, debugged and otherwise maintained in all future versions of the product. Once the users have seen it, they may grow accustomed to it, and so removing it from future versions may well be impossible. They may even like the feature so much that they begin requesting extensions and modifications to it, creating further burden on the development team. Frank justifies his actions to others in terms of providing value to users, and often professes a greater knowledge of the user demographic than what he actually possesses, so that he can claim how much the users will need a particular feature. But Frank's real motivations are not really about user satisfaction, but are about satisfying his own ego. Each new feature is an opportunity for him to demonstrate how clever he is, and how in touch with the user community.

Generic George

George delights in the design process. Pathologically incapable of solving just the immediate problem at hand, George always creates the most generic, flexible and adaptable solution possible, paying for the capabilities he thinks he will need in the future with extra complexity now. Sadly, George always seems to anticipate incorrectly. The castles in the air that he continually builds rarely end up with more than a single room occupied. Meanwhile, everyone must cope with the inordinate degree of time and effort that is needlessly invested in managing the complexity of an implementation whose flexibility is never required. It is a usual characteristic of George's work that it takes at least a dozen classes working together to accomplish even trivial functionality. He is generally the first to declare "Let's build a framework" whenever the opportunity presents itself, and the last to want to use the framework thus created.

Hacker Henry

Henry considers himself to be a true hacker – a code poet and geek guru. Still in the early stages of his career, he spends most of his life in front of a keyboard. Even when not at work, he is working on his own projects, participating in online discussion forums and learning about the latest languages and utilities. Software is his principal passion in life. This single-minded pursuit of technical knowledge has made him quite proficient in many areas, and has engendered a certain arrogance that generally manifests as a disdain directed towards those of his colleagues whom he regards as not being "true hackers". For his managers, Henry is a bit of a problem. They know that they can rely on him to overcome pretty much any technical challenge that might be presented to him, provided that the solution can be reached by doing nothing but coding. For unless it's coding, Henry's not interested. He won't document anything; certainly not his code, because he feels that good code is self-documenting. He is early enough into his career to have not yet been presented with the task of adopting a large code base from someone who subscribes to that same belief, and to have thereby seen the problems with it. Also, Henry can generally only be given "mind-size" tasks to do. His tasks have to be small and well defined enough for him to fit all their details in his head at once, as he simply refuses to write anything down. The architecture of enterprise-scale systems will likely forever be a mystery to him as he does not possess, and has no interest in developing, the facility with abstractions and modeling that is necessary to manage the design of large systems.

Incompetent Ian

Ian is a nice enough guy but is genuinely incapable of performing most of the job functions his position requires. It's not clear whether this is a result of inadequate education, limited experience or simply a lack of native ability. Either way, it is clear to anyone who works with Ian for any length of time that he is not really on the ball, and takes a very long time to complete even basic tasks. Worst of all, Ian seems to be blissfully unaware of his own incompetence. This can make for some embarrassing situations for everyone, as Ian's attempts to weigh in on technical discussions leave him looking naive and ignorant – which he also fails to notice. Ian tends to get work based upon his personable manner and the large number of friends he has working in the industry. Most of his employers have come to view him as a "retrospective hiring error".

Jailbird John

John has been working for his current employer a long time. A very long time. Longer than most of the senior management in fact. John has been working here so long that it is highly unlikely he will ever be able to work anywhere else. Over the years, his skill set has deteriorated so greatly and become so stale that he has become an entirely unmarketable commodity. He knows all there is to know about the company's legacy applications – after all, he wrote most of them. He has been keeping himself employed for the last decade just patching them up and making one piecemeal addition after another in order to try and keep them abreast of the business's changing function. Tired of chasing the latest and greatest technologies, he has not bothered learning new ones, sticking to the comfortable territory defined by the small stable of dodgy applications he has been shepherding for some years. John gets along with everyone, particularly those more senior to him. He can't afford the possibility of getting into conflict with anyone who might influence his employment status, as he knows that this will likely be the last good job he ever has. So he tries to stay under the radar, hoping that the progressive re-engineering of his pet applications with more modern technologies takes long enough for him to make it over the finish line.

Kludgy Kevin

Kevin is remarkably quick to fix bugs. It seems that he's no sooner started on a bug fix than he's checking in the solution. And then, as if by magic, the very same bug reappears. "I thought I fixed that", declares Kevin – and indeed he did – but not properly. In his rush to move on to something else, Kevin invariably forgets to check that his "fix" works correctly under some boundary condition or special case, and ends up having to go back and fix it again. Sometimes a third or even fourth attempt will be necessary. This is Kevin's version of "iterative development."

Loudmouth Lincoln

Terror of the cubicle farm, Lincoln incurs the ire of all those who sit anywhere near him, but remains blissfully unaware that he is so unpopular. His voice is louder than anyone else's by a least a factor of two, and he seems unable to converse at any volume other than full volume. When Lincoln is talking, everyone else is listening, whether they want to or not. People in his part of the office know a great deal more about Lincoln's personal life than they would like, as they have heard one end of the half dozen or so telephone calls that he seems to receive from his wife every day. Lincoln's favorite instrument of torture is the speakerphone. He always listens to his voicemail on speakerphone each morning, so that he can unpack his briefcase while doing so. He also likes to place calls on speakerphone so that his hands are free to type at his keyboard while conversing with someone else. He either doesn't realize or doesn't care that he is disturbing those nearby. Nobody seems to be game enough to tell him how inconsiderate he is being.

Martyr Morris

Morris is very conscious of the impression others form of him. Probably a little too concerned. He has observed that many of his colleagues associate long hours with hard work and dedication. The longer the hours, the harder you're working – and having a reputation as a hard worker can only be a good thing when it comes performance review time. So Morris makes sure he is at the office when his boss arrives of a morning, and that he is still working away when his boss leaves of an afternoon. Everyone agrees that Morris certainly puts in the hard yards, but are a little perplexed as to why his code is so often buggy and poorly structured. In fact, it seems like Morris has to put in extended hours in order to compensate for the poor quality of his work. The net result is that he gets almost as much achieved as his team mates who work more sensible hours. Morris hasn't yet twigged to the fact that his defect injection rate rises dramatically as he fatigues, meaning that the extra hours he works often have a negative effect on his productivity. Worse yet, his know-nothing manager rewards him for his dedication, thereby reinforcing the faulty behavior.

Not-Invented-Here Nick

Nick has an overwhelming drive to write everything himself. Due to hubris and ambition, he is rarely satisfied with buying a third party utility or library to help in his development efforts. It seems to him that the rest of the industry must be incompetent, for every time he looks to buy rather than build, he finds so many shortcomings in the products on offer that he invariably concludes that there's nothing for it but to write the whole thing himself. It also seems that his particular requirements are always so unique that no generally available tool has just the functionality that he needs. Not wanting to work inefficiently, he insists on only using tools that do exactly what he wants – nothing more, nothing less. Little wonder then that he finds himself having to write such fundamental utilities as text editors, file transfer programs, string and math utility libraries. The real problem is not that Nick's requirements are so unique, but that he deliberately fabricates requirements so specific that he can find commercial offerings lacking, and thereby justify reinvention of those offerings himself. In short, he is looking for excuses to write what he considers to be the "fun stuff" (the development tools) rather than the "boring stuff" (the application code). He generally has little difficulty in finding such justifications. Most people who work with Nick note with interest that the tools that he writes himself are rarely of the quality of the equivalent commercial offerings.

Open Source Oliver

Oliver is very enthusiastic about open source software development. He contributes to several open source projects himself, and tries to incorporate open source products into his projects wherever possible – and it's always possible; mainly because Oliver begins a project for the principal purpose of providing himself with an opportunity to try out the latest and greatest CVS build from Apache, Jakarta or wherever. Oliver rarely has to justify his technology selections to his colleagues, as he is always sure to surround himself with other open source believers. On occasions when he needs to explain the failure or buggy nature of some open source package, he relies upon the old saw "we can always fix it ourselves". However there never seems to be enough time in the schedule for this to actually occur; so every release of his project bristles with the underlying warts of its open source components. If all else fails, it can at least be said that the price is right.

Process Peter

If you want to see Peter get worked up, just start a discussion with him about the poor state of software development today. He will hold forth at length, and with passion, on where it has all go wrong. And Peter has decided that all of software's woes have a common genesis – a lack of disciplined process. Peter's career history reads like a marketing brochure of process trends. BPR, Clean Room, Six Sigma, ISO – he's been a wholehearted enthusiast of them all at one time or another. His dedication to strict process adherence as a panacea to a project's quality ills is absolute, and he will do almost anything to ensure that ticks appear in the relevant boxes. Unfortunately, this uncompromising approach is often self-defeating, as it denies him the flexibility to adapt quality levels on a case-by-case basis. It has also made him more than a few enemies over the years. He is prone to considering the people component of software development as a largely secondary consideration, and views programmers a little like assembly line production workers – interchangeable parts whose individual talents and proclivities are not so important as the procedures they follow to do their work. Those subject to such views tend to find it more than a little dehumanizing and impersonal.

Quiet Quincy

Quincy is one of those guys who has no need to brag about his technical skills or the depth of his technical knowledge. He's not much interested in being "alpha geek" at the office, he just wants to do a good job and then go home to his wife and children. Quietly spoken and unassuming, he looks on with amusement at Zealous Zack's ever-changing enthusiasms and shakes his head, knowing that in a few more years Zack will have gained enough experience to know that the computing industry is full of "next big things" that generally aren't. Given a task, he just sits down and does it. He doesn't succumb to heroic bug-fixing and late night coding efforts – his code is good enough to begin with that there are rarely any problems with it. He probably won't get many pats on the back from management, whose attention will largely be captured by the technical prima donnas that swan around the project space, dropping buzzwords and acronyms like they were the names of celebrities they knew personally. But without Quincy and those of his ilk, the project would fail – because someone has to get the work done.

Rank Rodger

Rodger is very good at what he does. He's a techie through and through, and delights in problem solving. The problem is that Rodger lives in his head. At times he feels like a brain on legs, so focused is he upon intellectual pursuits. His body is a much neglected container for cortical function that he generally pays little attention to, except to meet its basic functional requirements for food and clothing. As a result, there is a certain funk surrounding Rodger which nearby colleagues are all too aware of, but of which Rodger is olfactorily ignorant. Halitosis is his constant companion and dandruff a regular visitor. In general, he has unkempt appearance – his shirt often buttoned incorrectly, hair not combed and tie (which he wears only under the greatest duress) knotted irregularly. Rodger doesn't really care what others think of him and is largely unaware of the message his poor grooming and hygiene is sending to others. Rodger is likely to remain unaware for a long time, as nobody can think of a way of broaching the topic with him that wouldn't cause offense.

Skill Set Sam

Sam is just passing through. If he is a contractor, everyone will already be aware of this. If he is permanent staff, his colleagues might be a little surprised to know just how certain he is that he won't be working here in a year's time. Sam is committed to accumulating as much experience with as many technologies as he possibly can, in order to make himself more attractive to future employers. His career objective is simply that he remain continually employed, earning progressively higher salaries until he is ready to retire.

Toolsmith Trevor

Trevor loves to build development tools. He can whip you up a build script in a few minutes and automate just about any development task you might mention. In fact, Trevor can't be stopped from doing these things. He is actively looking for things to automate – whether they need it or not. For some reason, Trevor doesn't see the writing of development tools as a means to an end, but an end in itself. The living embodiment of the "Do It Yourself" ethic, Trev insists on writing common development tools himself, even if an off-the-shelf solution is readily available. Rather than chose one of the million commercially available bug tracking applications, you can rely on Trevor to come up with an argument as to why none of them are adequate for your purposes, and there is no solution but for him to write one. At the very least, he will have to take an open source tool and customize it extensively. So too with version management, document templates and editor configuration files. Trevor is right into metawork, with the emphasis on the meta.

Unintelligible Uri

English is not Uri's native tongue. This is blatantly obvious to anyone who attempts to communicate with him. He speaks with a thick accent and at such a rapid pace that listeners can go several minutes in conversation with him without having a clear idea of what he has said. Trying to work with Uri can be an excruciating experience. He cannot contribute to technical discussions effectively, regardless of how well informed he might be, because he is always shouted down by those with more rhetorical flair, regardless how uninformed they might be. Delegating work to him is a dangerous undertaking because you can never be certain that he has really understood the description of his assignment; he tends to respond with affirmative clichés that can be easily said, but don't necessarily reflect that information has been successfully communicated. Very often, people choose simply not to bother communicating with Uri, because they find it both exhausting and frustrating. Whoever hired Uri has failed to appreciate that fluency in a natural language is worth ten times as much as fluency in a programming language.

VB Victor

Sometime in the nineties Victor underwent what is colloquially referred to as a "Visual Basic Lobotomy". He found himself a programmer on a misconceived and overly ambitious VB project, and fought to write a serious enterprise application for some years in a language that was never conceived for more than small scale usage. Visual Basic Land is a warm and soothing place, and Victor let his skill set atrophy while he slaved away at VB, until eventually VB was all he was good for. Now, dispirited and deskilled, he is a testament to the hazards of building your career upon a narrow technological basis. Victor will likely survive a few more years, pottering from one VB project to the next, until he loses the enthusiasm even for that.

Word Salad Warren

Unlike Uri, Warren's native tongue is English; but it does him little good. Listening to Warren explain something technical is like listening to Dr Seuss – all the words make sense when taken individually, but assembled together they seem to be mostly gibberish with no coherent message. Such is Warren's talent for obfuscation, he can take simple concepts and make them sound complex; take complex topics and make them sound entirely incomprehensible. This is big problem for everyone attempting to collaborate with Warren, for they generally find it impossible to understand the approach Warren is taking in solving his part of the problem, which virtually guarantees it won't work properly in conjunction with other's work. On those rare occasions when he tries to document his code, the comments aren't useful, as they make no more sense than Warren would if he were explaining the code verbally. Management has made the mistake of assuming that Warren's diatribes are inscrutable because he is so technically advanced and is describing something that is inherently complex. That's why he is in a senior technical position. But his pathetic communication skills are a major impediment to the duties he must perform as a senior developer, which routinely involve directing and coordinating the technical work of others by giving instructions and feedback. Warren is a source of great frustration to his colleagues, who would give anything for precise and concise communication.

X-Files Xavier

Xavier takes a little getting used to. Although his programming skills are decidedly mature, his personality seems to be lagging behind. He has an unhealthy fascination with Star Trek, Dr Who and Babylon 5. Graphic novels and Dungeons and Dragons rule books are littered about his cubicle, and he can often be found reading them during his lunch break, which he always spends in front of his computer, surfing various science fiction fan sites and overseas toy stores. Project meetings involving Xavier are generally ... interesting, but somewhat tiring. He regularly interjects quotations from Star Wars movies and episodes of Red Dwarf, laughing in an irritating way at his own humor, oblivious to the fact that others without his rich fantasy life are not amused by his obscure pop culture references. Xavier seems to spend most of his time by himself. No one has ever heard him mention a girl-friend. Those who have worked with him for any length of time know that he is best kept away from customers and other "normal people" who would not understand his eccentricities.

Young Yasmin

Yasmin has only been out of University for a few years. She is constantly surprised by the discrepancy between what she was taught in lectures and what actually appears to happen in industry. In fact, there seems to be a good deal that happens in practice that was not anticipated at all by her tertiary education. She concludes that the numerous shortcuts, reactive management and run-away bug count of her projects are just localized eccentricities, rather than a widespread phenomenon. Yasmin fits well into the startup company environment, with its prevailing attitude of "total dedication." Indeed, she is the target employee demographic of such firms. She is at that stage of life where she has the stamina to work 60 and 70 hour weeks on a regular basis. She is not distracted by family commitments, and is ambitious and eager enough to still be willing to do what is necessary to impress others. Lacking industry experience and the perspective that comes with maturity, she is not assertive enough to stand up to management when they make excessive demands of her.

Zealous Zack

Zack is a very enthusiastic guy. In fact, there seems to be very little going on in the world of computing that Zack is not enthusiastic about. Like a kid staring in the candy store window, Zack gazes longingly at every new buzzword, acronym and advertising campaign that crosses his path, immediately becoming a disciple of every new movement and technology craze that comes along. Sometimes these enthusiasms bring with them certain ideological conflicts, but Zack is too busy downloading the Beta version of the next big thing to be worried about such matters. He runs Linux on his home PC, has a Mac Mini in his living room, and worships at the church of Agile. Having Zack on your project can be challenging, particularly if he exercises any control over technology selection. He will invariably try and load down your project with whatever "cool" technologies he is presently over-enthused about, and delight in the interoperability problems that result as an opportunity to introduce even more technologies to save the day. Zack never quite learnt to distinguish work from play.

---

[*] First published 24 Jan 2006 at http://www.hacknot.info/hacknot/action/showEntry?eid=81

The Hazards of Being Quality Guy*

Perhaps you've seen the Dilbert comic about Process Girl. At a meeting, the Pointy Haired Boss introduces Process Girl as "the one who has the answer to everything", at which point Process Girl chimes in parrot-like with "Process!" She then denounces the meeting as inefficient because the participants have no process to describe how to conduct a meeting. By a unanimous vote she is expelled from the meeting. As he escorts her out of the room, Dilbert offers by way of consolation "at least you lasted longer than Quality Guy."

And now I must reveal a shocking truth ... ladies and gentlemen (rips open shirt to reveal spandex body suit with "Q" emblazoned on the front) ... **I** am Quality Guy. I am that much maligned coworker that you love to hate. I am your local ISO champion, the leader of the Software Engineering Process Group and the mongrel who overflows your inbox with links to articles about process improvement. I'm the trouble maker that asks embarrassing questions in meetings like "why aren't we doing code reviews?" and "where's the design documentation?" I am the one that dilutes your passionate discussions on J2EE and SOAP with hideously unfashionable prattle about CMM and the SEI.

And like my namesake in the Dilbert comics, I am ostracized by my peers and colleagues. I am renounced as being a "quality bigot" and dismissed as impractical and too focused upon meta-issues to actually achieve anything worthwhile. I am perceived as an impediment to real work and characterized as a self-righteous, holier-than-thou elitist. My suggestions of ways to improve my team's work habits are interpreted as personally directed criticisms and thereby evidence that I am "not a team player".

From my point of view at the periphery of the team, the earnest activity of you and your geek friends seems somewhat farcical. You seem to be perpetually distracted by the shiny new technology toys that the vendors are constantly grunting out. You are hopelessly addicted to novelty and consumed by the frenetic pursuit of the latest bandwagon. You seem to be entirely unconcerned that "beta" is synonymous with "buggy" and "new" with "unproven". The projects of my successive employers march by me like a series of straight-to-video movies, each baring the same formulaic plot wherein only the names of the participating technologies have been changed to protect the innocent. I feel compelled to yell out "stop!", "think!" and "why?", but it is hard to be heard when you're in geostationary orbit around Planet Cool and in space, no one can hear you scream.

Friends, this is what it is to be Quality Guy, and it ain't no party.

If you think you or a loved one might be in danger of becoming a Quality Guy sidekick, let me offer you this one piece of advice – never reveal your true identity to your coworkers. It is a sure recipe for alienation and isolation. Keep your shirt closed to the top button, so that your superhero garb will go unnoticed. Eschew all quality-related terminology from your public vocabulary and substitute terms from the jargon file[1]. Hide any books you might have that do not relate directly to a technology.

When it comes to development practice, with a little ingenuity you can institute a number of quality-related practices within the sandbox of your own development machine, without needing to reveal to others that your sphere of concern extends beyond the acronymic:

- If you find yourself in an environment without version control, install a free version control system such as CVS or CS-RCS on your own machine. You can at least maintain control over those files that you are immediately involved with.
- If there is no prevailing coding standard, employ one for your own code without revealing to others that there is any guiding hand of consistency in your code (that would be un-cool).
- If there is no unit testing, write your own in a parallel source tree visible only to yourself using the free `xUnit` package appropriate to your platform.
- If there is no design documentation, reverse engineer the existing code into some hand-drawn UML diagrams and then stash them away where others won't find them, keeping them just for your own reference.
- No requirements? Start your own mini-requirements document as a local text file, and question the developers and senior team members around you to try and flesh it out. You can at least try and restrict uncertainty with regard to your own development objectives.

Remember, the secret to surviving as a Quality Guy is to keep your true identity a closely guarded secret. That way you can still be one of the gang and remain non-threatening whilst still being able to take some satisfaction from the limited degree of quality enforcement you can achieve through isolated effort.

[*] First published 3 Sep 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=20
[1] http://www.jargonfile.com/

A Dozen Ways to Sustain Irrational Technology Decisions*

External observers often think of programmers as being somewhat cold and emotionless. Because our day-to-day activities are largely analytical in nature, it has become a part of the developer stereotype that we are dispassionate and rational in our manner and decision making. Those who have watched programmers up close for any length of time will know that this is far from the case. I believe that emotion plays a far larger part in IT decision making than many would be willing to admit. Frequently developers try and disguise the emotive nature of their thinking by retrospectively rationalizing their decisions, but not being well-skilled in interpersonal communication, are often unconvincing. If you've ever witnessed or taken in part in a technological "holy war", then you'll already have witnessed the unhealthy way that stances held by emotional conviction can be misrepresented as being the result of rational analysis.

The Causes

Novelty

The majority of irrational technical selections I've seen have their origin in a senior techie's fascination with a new technology. For an uncommon number of developers, the lure of an untried API or the novelty of a new development model is simply irresistible. Such folks seem to be focused on the journey rather than the destination – which is philosophically delightful but practically frustrating. The urge to play with a new toy seems to overwhelm the ability to rationally evaluate a technology on its merits, as if it's "newness" excused any faults and weaknesses it might have. There seems to be a strong "grass is greener" effect at work here. The weaknesses of existing technologies are known because they have been teased out by the development community's experience with it. But a new technology has an unblemished record. The absence of community experience means that no one has encountered its inevitable flaws, or pushed the boundaries of its capabilities. Psychologically, it is easy to be drawn to the new technology based on the implied promise of perfection, as compared to the manifest imperfections of current technologies.

Ego

Programmers are not a group lacking in self-confidence; at least when it comes to technical matters. In fact, the intellectual arrogance of some can be quite stunning. For those with decision-making authority, the burden of ego can be a substantial liability. A technology selection based solely upon technical merit is easily defended by dispassionate reference to facts, but once the outcome is identified with the individual who made it, ego comes into play. Any challenge to the decision tends to be interpreted as a challenge to the authority of the decision maker. Any criticism of the selected technology tends to be emotionally defended, because the party who selected it feels that fault is being found with them personally. They are likely also sensitive to the potential for injury to their image and reputation that might come from being responsible for a poor technology decision. It is difficult to retain status as the alpha geek when you are known to have made poor technical decisions. Managers, in particular, are acutely aware of the way their behavior and ability is perceived by others. Having been drawn in by the false promises of glossy product brochures, the misinformed technical manager is poorly positioned to subsequently defend technology decisions. Such managers are frequently those to be found most passionately and aggressively defending their decisions.

Fashion

An alarming number of developers seem to be slaves to technical fashion. Plagued by a "gotta get me some of that" mentality, the arrival of almost any new product or development tool is accompanied by an almost salivatory response. They rush to evaluate the new offering and to share their experiences with like-minded others who also like to be at the leading edge. These programmers fit well and truly into the "early adopter" category, or as I like to call them "crash test dummies." Like their mannequin counterparts, they are forever running head long into collisions – in this case, with technologies. By observing the results, the rest of us can learn from their often hard-won experiences, without having to suffer the frequent injuries that tend to result.

Ideology

As frequent as it is unrecognized, ideological conviction seems to be a major driver behind many technology decisions. Many developers remain convinced that open source software will save the world, enable black and white peoples to live in racial harmony, cure cancer and eliminate hunger and poverty. They may be right, but none of these are rational reasons to select a particular offering over a proprietary alternative for a particular commercial application. But for many, it is automatic and unquestioned that open source software is the way to go, as a matter of moral imperative, regardless of the merits or otherwise of that software.

The Techniques

Once the commitment to a particular technology has been publicly made, its proponents must then be prepared to defend their decision in the light of any negative development experience. If the technology was selected for irrational reasons, then those identified with its selection must now become apologists for the technology, seeking to minimize and quash any information that might reflect poorly on the technology and transitively, upon themselves.

Here are twelve techniques I have seen used to sustain a bad technology decision in the face of experience that puts that technology's selection in doubt.

1. Deny That Negative Experiences Exist

This is a common technique amongst the "kick ass" school of management. When faced with evidence that casts your technology selection in an unfavorable light, simple deny that the evidence exists. Even if someone can demonstrate to you first hand the problems that have been encountered, you can employ a "shoot the messenger" approach to distract attention away from the evidence being presented, and put the messenger on the defensive. You will need to be in a position of sufficient authority, and surrounded by suitably spineless colleagues, to make "black is white" declarations hold fast and create a localized reality distortion zone. It may sound fantastic, but in practice it is quite common for authority to usurp reality.

It is not a technique unique to the IT profession. In his memoirs "Inside the Third Reich", Albert Speer relates a situation in which Hermann Göering employed exactly this technique. When Göering was advised that American fighters had began to encroach upon German skies, he refused to accept the report, despite being presented with irrefutable evidence by one of his generals. He simply issued an official order stating that nobody had seen any fighters.

2. Claim "We'll Fix It Ourselves"

When an open source product is selected but ultimately found wanting, the "we can fix it ourselves" apology is often the first one that is trotted out. The availability of the source code means that you can ostensibly patch the product yourself, submit that patch to the open source project, and then carry on. Whenever a colleague finds a bug in the technology, just dismiss their complaints with the directive to "just fix it yourself", and the problem will go away ... for you, anyway.

3. Claim That Bugs Are Intellectual Property

This is a sneaky but effective one. Make it known to your colleagues that they cannot report any problems they find with the new technology to the vendor (or the community, in the case of open source software) as that would equate to divulgence of information that has been gathered at company expense. In the strictest sense, the knowledge of the bug's existence is the company's intellectual property. Exactly what kind of intellectual property it is, is open to question. It could be "confidential", but it seems doubtful that it is of enough significance to possess the necessary "quality of confidence". In any case, it doesn't really matter. You can rely upon others being sufficiently intimidated by the implied threat of prosecution for IP infringement to remain silent.

4. Claim "It Will Be Fixed In The Next Release"

This piece of misdirection can be used to postpone problems almost indefinitely. It is particularly handy for products that are on a short release cycle, as the promise of a fix is always just around the corner (and with it, the potential for the introduction of new bugs – but ignore that). If the bug is not actually fixed in the next release, then it's hardly your fault. Blame the vendor, blame the development community, lament the state of software development in general ... do anything to divert attention away from the original source of the technology's selection.

5. Make The Bug Reporting Process Unwieldy And Onerous

A worthwhile bug report takes a bit of effort to produce. Sample code, screenshots and instructions to reproduce the buggy behavior are all part of a conscientiously compiled bug report. But if that is all that is required, there will be some developers willing to take the time to write them. You can make the lodging of a bug report more daunting by requiring developers to lodge an entire specification of the desired (non-buggy) behavior, including requirements, a mock-up or prototype, design specification and test specification. This can take days. They'll quickly learn that it's simply not worth the effort to report bugs via such a lengthy process, and to move directly from discovery of a bug to the search for workarounds or alternative approaches.

6. Claim "It Works For Me"

An indirect form of denial exists in claiming that you have been unable to reproduce the bug yourself, so the complainant must be doing something wrong. Due to the almost unlimited potential for interactions between software components, libraries and operating system functions, it is easy to simply point somewhere in the direction of this programmatic thicket and declare "the problem's probably in there."

7. Appeal To Non-Quantifiable Benefits Yet To Be Realized

If enough difficulties are encountered with your chosen technology, it's only a matter of time until someone starts suggesting alternatives. When your opponents open fire with the feature list of their favorite competing technology or product, you need a reply. It is best to appeal to non-quantifiable and non-functional benefits as it is impossible to prove that they have not been realized. "Flexibility" and "maintainability" are a few non-functional favorites that you can claim are being realized by your technology selection, regardless of what the reality may be.

8. Employ The Power Of Standards

A technology that has been embodied in a standard already has a significant head start on non-standardized competitors. If the standard is one that has been accepted by major vendors as a basis for their own product offerings, then all the better. The psychological principal being appealed to here is that of "social proof" - the belief that popularity is indicative of worth. Indeed, widespread acceptance of a standard (or a technology implementing a standard) is unlikely to occur if the notion is completely without value, but there is no guarantee of you achieving the same success in your own context as others have achieved in theirs. However, many will ignore the need to consider application-specific issues in deciding the merit of a technology. If IBM, Microsoft or some other big name says it's good, then it must be good - for everyone, all the time, regardless of what the constraints of their particular problem may be. To appreciate how seductive this faulty reasoning can be, consider how many times you've seen a J2EE application that was written simply for the sake of using J2EE, even though there was no real need for a solution with a distributed architecture.

9. Maximize Investment

One of the best ways to get a technology on a solid foothold in your organization is to maximize your investment in it as quickly as possible. This can be achieved by forward-scheduling tasks that use the technology the most, so that the number of hours invested in using it accrue quickly. You might justify this by presenting the host project to management as a "pilot" of some sort, where the technology is being evaluated on its merits. But so long as you can silence any negative findings that might emerge from that ersatz "evaluation", you are also strengthening the project's commitment to the continued use of that technology. What project wants to incur the schedule burden of having to swap technologies and re-implement those parts of the project based upon the now defunct technology? If you can just suppress criticism for long enough, the project will soon reach a point of no return, after which it becomes infeasible to make technology changes without incurring an unacceptable schedule penalty.

The bigger a company's financial investment in a technology, the more reticent it will be to discard it. So you will find it easier to keep expensive technologies in use. You can increase expenditure by purchasing entire product suites, or choosing products so complex that you can justify hiring highly paid consultants to tailor them to your project environment or teach your staff how to use them. Once all that time and money has been invested, it will become extremely difficult for anyone to abandon the technology due the financial inertia it has acquired.

10. Exclude The Technically Informed From The Decision Making

As a self-appointed evangelist for your chosen technology, your worst enemy is the voice of reason. The technology's inability to fulfill the promises its vendor makes should be no obstacle to its adoption in your organization – and indeed, it won't be, so long as you can keep those who make the decisions away from those who know about the technology's failings. Let their be no delusion amongst your staff and colleagues that it is management's purview to make these decisions, and the techie’s job to implement their decision. Some will try and argue that those who know the technology most intimately (technical staff) are in the best position to judge its value. Assure them that this is not so and that only those with an organizational perspective (management) are in a position to assess the technology's "fit" with the corporate strategy. Allude to unspoken factors that influence the decision to use this technology, but are too sensitive for you to discuss openly (conveniently making that decision unassailable).

11. Sell The Positives To Upper Management, Hide The Negatives

Question: How does a fish rot?

Answer: From the head down.

If you can get those in senior management to develop some identification with the technology then you will have made some powerful allies. Assuming they are technically uninformed, make your management a sales pitch for the technology in which you emphasize all the positives and completely neglect the negatives. Give them glossy brochures advocating the technology, and appeal to their competitiveness by providing testimonials from big-name managers, as if to suggest "this technology is what the best managers are getting behind"; the implication being that your own management are not amongst "the best" unless they follow suit. The ego-driven push from above is almost impossible to counter with a factual push from below. Authority trumps reason in many organizations.

12. Put A Head On A Pike

It is part of the barbarian tradition to place a head on a pike at the entrance to your domain, to warn those approaching of the fate that awaits them if they don't follow the rules. It's crude, but undeniably effective. Actual decapitation is frowned upon in most office environments, but you can still put a figurative "head on a pike" to make it clear to others that dispute over your chosen technology will not be tolerated. If you have the authority, firing someone who expresses a dissenting opinion should be adequate to ensure the remaining staff fall into line. Otherwise, some form of public humiliation – a verbal dressing down in a common area of the office, for instance – will have to do. In either case, it is important that you adopt some pretense for your actions that is not directly related to the issue of technology selection. Unfair dismissal laws being what they are, you need to be a bit careful here. Witnesses will know, however, from the greater context that the real reason for this retribution is the target's opposition to the technology decision you made, and will make a note to themselves not to express their own concerns about the technology, lest they also be made an example of.

Conclusion

IT managers, developers and other technical staff are no less susceptible to self-deception and political ambition, simply because they work in a field in which analytical thought is traditionally valued. When it comes to the selection of a technology from a field of competitors, the complexity and number of factors to consider often leads to a tendency to abandon detailed, rational analysis and make decisions on an arbitrary, emotive basis. If the technology selected fails to live up to its promise, those who selected it then face the difficult task of rationalizing its continued usage, lest their decision be overturned and they lose face as a result. By employing one or more of the techniques identified above, a skilful manager or senior technician can avoid this embarrassment and force the continued usage of an unsuitable technology, while they work by other means to distance themselves from the original decision.

[*] First published 5 Oct 2005 at http://www.hacknot.info/hacknot/action/showEntry?eid=79

My Kingdom for a Door*

"All men's miseries derive from not being able to sit in a quiet room alone." – Blaise Pascal

In some interviews there comes a point where you realize that you don't want the job. It might be the moment you discover that the employer has conveniently omitted from the published job description the requirement for the incumbent to spend 50% of their time maintaining a one million line legacy application, written in Visual Basic. It may be shortly after you state your salary expectation, only to be greeted with a look of blank astonishment. For me, it is often the point at which the interviewer reaches into their bag of interview clichés and asks a question so trite that it betrays the total absence of advance preparation and original thought. Once the role has been safely relegated to the "no thanks" pile, it is difficult to resist adopting a certain playfulness while waiting out the duration of the interview, as courtesy demands.

For example, when asked "Where do you see yourself in five years time?" I like to borrow a witticism from comedian Steven Wright, and respond "I don't know – I don't have any special powers like that." If asked "Why are manhole covers round?" I might reply "Because God made them that way", simply to see if they will dare broach a topic traditionally considered taboo in interviews. And if they should enquire "What are your career goals?" I will almost certainly reply "I have only one – I want a door."

But in this last I'm only partially being facetious, for one of the most consistently difficult aspects of every software development effort I've been a part of has been the physical environment in which it is conducted. Having abandoned the lofty career goals of my youth (such as producing quality software) I have deliberately set my sights a little lower. These days, my sole ambition is to have an office with a door. My professional nirvana would then be to close that door, so I can get on with my work undisturbed.

As challenging as technical issues can be, they are at least considered approachable by most organizations. But environmental problems, particularly noise levels, seem to universally receive short shrift, and are often dismissed as an unfortunate but unavoidable part of office life and beyond anyone's ability to deal with.

Of course, the problem of office noise is far from intractable. Numerous approaches can be taken to relieve or at least ameliorate the problem, the most obvious of which involves the reintroduction of an antiquated and long neglected piece of spatial division technology – the door. The real reasons that environmental issues go unattended are somewhat different.

Brain Time Versus Body Time

Software developers are knowledge workers. Our job is to produce intellectual property. You would think it self-evident that work of this nature requires sustained concentration, and that it is easier to concentrate when things are quiet.

Back in my school days, these facts seemed to be widely known and accepted. When you went to the library, the school librarian (who, in my school, was a particularly ferocious woman the students referred to as "Conan The Librarian") would do her best to see that the library was quiet. Why? *Because people were trying to study, to think, to concentrate.* When there was an exam to be done, the exam would be conducted in complete silence. Why? Because it's easier to concentrate on your exam when it's quiet. When the teacher gave the class time to work on an assignment, the class was expected to be silent. Why? Because it's easier to think about your assignment when it's quiet.

In university too, there was little dispute about the necessity for a quiet environment when doing intellectual work. The libraries and exam halls are silent, the lecture theaters and tutorial rooms are quiet so that the speaker may be heard and their message understood.

Prior to entering the workforce, I thought nothing of it. It all seemed to be just common sense. Imagine my surprise then to discover that the corporate world had decided that none of it was true. That, in fact, you don't need quiet in order to concentrate effectively – you can work just as well when immersed in an environment that is a noisy as your local shopping center. Or so I infer is the reasoning, because the standards in both office accommodation and behavior seem to have been determined with such an assumption in mind.

Sitting at my desk at work, I am surrounded by distraction and diversion, which everyone just seems to accept will not impair my ability to work at all. But my own impression is very much to the contrary. I find myself constantly frustrated and annoyed by the ceaseless chatter around me and the incessant whir of printers and photocopiers. I have never known a workplace to be any different.

How is it that the corporate and academic worlds seem to have completely different ideas about what characterizes an environment conducive to intellectual activity? Why is it that the academic community seems to have got it right, and the corporate community ubiquitously has it wrong? Surely employers are not knowingly paying their staff to be only semi-productive, are they? Unless the corporate world is consistently behaving in a self-defeating and irrational way, I must simply be mistaken about the effect this office noise is having on me.

Perhaps I am actually quite unaffected by the conversations that my cubicle neighbors are having, on matters unrelated to my work ... all day. Perhaps the four foot high partition which separates me from them is actually enough to reduce their inane chatter and laughter to a distant whisper – I guess the sound dampening cloth on it must have some effect. Although the partition only covers two of the four sides of my "cubicle", perhaps adopting a "glass half full" attitude would make the lack of privacy less disturbing. Perhaps the sound of the printers and copiers in the facilities area, just three feet away from my desk, really isn't that loud. Perhaps the guy in the next cubicle who insists on checking his voice mail through the speakerphone isn't the sociopath he appears to be, and I'm just not sufficiently tolerant of others. Perhaps it's not really all that visually distracting to have people walking through the corridor beside my cubicle every few minutes. Maybe some blinkers, like those given to cart horses, would lessen the effect of constant movement at the periphery of my vision. And perhaps the ten mobile phone calls that my surrounding cubies seem to get every day, each one heralded by a distinctive and piercing ring-tone sampled from some Top 10 dance hit, really isn't as wearing as what I think it is. And maybe having a pair programming partner leaning over your shoulder, barking in your ear and correcting your every typographic error isn't an obnoxious novelty that removes what little remaining chance there is of thoughtful consideration occurring in the modern workplace, but a mechanism for solving complex problems by having a chat over a nice cup of tea.

Or perhaps, just perhaps, the cubicle farm is a fundamentally unsuitable work environment for software developers. But how could that be, when the "open plan" office is the corporate norm? Could organizations really be so blind as to routinely give their staff an environment which is not conducive to the conduct of their work?

How could such a patently irrational trend develop and persist?

It's About Money

The modern cubicle had its genesis in 1968, when University of Colorado fine-arts professor Robert Propst came up with the "Action Office" – later commercialized by Herman Miller[1]. At the time, offices usually contained rows of desks, without any separation between them. At least cubicles were an improvement. But once the facilities management people cottoned onto the idea of putting people in boxes, their focus became achieving maximum packing density and consideration of noise and interruption went out the window (if you could find one). That mentality persists today, largely because the costs associated with office accommodation and office space rental are concrete expenditures that appear on a balance sheet somewhere. Somebody is accountable for those costs, and therefore seeks to minimize them. But the costs of lost productivity due to an unsuitable work environment aren't readily quantified, they just disappear "into the air", and so are easily forgotten or disregarded. There are also tax breaks in some localities, where legislation exists making it quicker to write off the depreciation of cubicles more quickly than traditional offices.[2]

It's About Rationalization

The ostensible benefits of an open-plan office are its moderate cost, flexibility, facilitation of teamwork and efficient use of space. These are the attributes by which cubicle systems are marketed[3]. Note that the ability to create an environment suitable for knowledge workers is not amongst those features.

Flexibility, although a possibility, is seldom realized in IT-centric environments where the need to re-route power and network cabling makes people reticent to re-arrange cubicles to any significant extent. Even individual variation and customization is discouraged in many workplaces, where such non-conformity is viewed as a threat to the establishment.

It is also commonly held that cubicles "promote communication" amongst staff. Unfortunately, one man's "communication" is another man's "distraction", the difference being whether the desire to participate is mutual. Alistair Cockburn, never one stuck for a metaphor, describes the wafting of conversation from one cube to the next as "convection currents of information"[4] and promotes the benefits that might arise from incidental communication. But when one is trying to concentrate, these currents of information become rip-tides of noise pollution that one cannot escape. The result is frustration and aggravation for the party on the listening end.

Unsurprisingly, companies that produce modular office furniture claim that cubicles are fabulous, and choose to selectively ignore their manifest disadvantages. In the advertising literature[7] for their "Resolve" furniture system, Herman Miller lauds the necessity of teamwork:

All the accepted research in this field says you have to have more visual and acoustic openness to get the benefits of a team-based organization.

... and downplays the need for individual work:

Although there will always be types of work that require intense concentration and protection from distraction, our research suggests that these needs can be effectively met outside assigned, enclosed workstations – through remote work locations or on-site, shared, "quiet rooms" for instance.

In other words, the workplace should be optimized for collaborative work, and those who want to concentrate can go elsewhere. Indeed, it seems to be a growing misconception amongst designers and managers that a high level of interaction and collaboration is a universal good, the more the better, and that the downsides don't matter.

For knowledge workers, who spend the vast majority of their time in isolated contemplation, this is decidedly bad news. Those who fit out offices seem to be either gullible enough to believe glib rhetoric such as the above, or more likely, choose to remain willfully ignorant of the fundamental requirements of their staff. Herman Miller would have you believe that the cubicle environment is good for your software development effort as well:

But the benefits of physical openness are gaining recognition even among the "gold-collar" engineers and programmers of Silicon Valley.

"The programming code we write has to work together seamlessly, so we should work together seamlessly as well", says a Netscape Communication programmer and open-plan advocate quoted recently in the New York Times.

Clearly, it is inane to suggest that software can be invested with desirable runtime behavior by adopting parallel behavior in the team that develops it. Does the code execute more quickly if we write it more quickly? Will it be more *user* friendly if the developers are more friendly toward each other? No – it is just nonsensical wordplay. But the use of such faulty "proof by metaphor" techniques is illustrative of how desperate the furniture industry is to ignore the workplace realities they are producing, and the superficial level of thought that they employ in promoting their ostensible success.

Consider the following statement, again from Herman Miller:

> *Recent studies also indicate that people become habituated to background office noise after prolonged exposure. Over time, people get used to the sounds of a given environment, and noises that initially have a negative impact on performance eventually lose their disruptive effect.*

Or perhaps, workers simply give up on the issue of office noise after their prolonged attempts to deal with it are continually met with stonewalling and denial. No references are given, so it is impossible to gauge the validity or relevance of these studies. But it sounds so inconsistent with known research in this area that one cannot help but be suspicious.

Many studies have examined the effect of background speech on human performance.[5] One phenomena that consistently recurs is the "Irrelevant Speech Effect" (ISE). In ISE experiments, participants are given tasks to do while being subject to speech that is unrelated to the task at hand. Susceptibility to ISE varies between individuals, but in general ISE is found to be "detrimental to reading comprehension, short-term memory, proofreading and mathematical computations."[6] In general, work that requires focus and ongoing access to short-term memory will suffer in the presence of ISE and other distractions and interruptions.

It's About Status

Real estate has always been an indicator of status. Whether you're a feudal lord or a middle manager, the area in your command is usually proportional to your perceived status and importance. Those who suggest that the cubicle is an unavoidable part of the office landscape are often those whose status precludes them from ever having to occupy one, and who have a vested interest in the distribution of office space remaining exactly as it is – in their favor. The unstated purpose of the cubicle is to serve as a container for the "have-nots", to more obviously distinguish them from the "haves." The preoccupation with offices (and the number of windows therein) and car parking spaces is often quite baffling to techies, who think first in terms of utility rather than perception. But for those more "image oriented," the true worth of corporate real estate has nothing to do with functionality and everything to do with positioning.

Float Your Mind Upstream

I would like to be able to say that companies are gradually realizing that knowledge workers such as software developers need support for both team interaction and distraction-free individual work, and are making changes to workplace accommodation accordingly. But I would be lying.

In truth, the workplace's suitability as a place to work is likely to sink below even its currently deplorable standard. The trend is towards ever smaller cubicles with fewer and lower dividing partitions. A 1990 study by Reder and Schwab found that the average duration of uninterrupted work for developers in a particular software development firm was 10 minutes. That's revealing, because it generally takes about 15 minutes to descend into that deep state of contemplative involvement in work called "flow". During the period in which one is transitioning to a state of flow, one is particularly sensitive to noise and interruption[7]. If you're interrupted every 10 minutes or so, chances are you spend your day struggling to focus on what you're doing, being constantly prevented from thoughtful contemplation of the problem before you by visual and auditory distractions around you ... and that's the typical working day of many software developers. As DeMarco and Lister comment "In most of the office space we encounter today, there is enough noise and interruption to make any serious thinking virtually impossible." With the addition of some doors into the environment, developers could at least control their noise exposure.

Look around you now, and what do you see? Chances are there will be at least one and probably many of your colleagues wearing headphones. It's common practice for software developers to retreat into an isolated sonic world as the only way they have of overcoming the incessant distraction around them. Some companies pipe white noise into individual cubicles to try and mask the surrounding noise. I've found it helpful to run a few USBpowered fans from my computer – their quiet hum serves much the same purpose, as well as compensating for the often inadequate air conditioning.

Why don't developers revolt? Why is it so rare to hear them vocalize their complaints? Talk to them in private and they'll likely concede that their work environment is too noisy to enable them to work effectively. But they're unlikely to make those concerns public, for fear of retribution or simply because they know that the noise level will be dismissed as being an inherently intractable problem.

So we will continue to grind our teeth and shake our heads in disbelief while listening to the dull roar of the combined efforts of the printers, fax machines, photocopiers, telephones, speakerphones, inconsiderate coworkers, slamming doors, hallway conversations immediately beside our desks and wonder how we can be expected to work effectively amidst such a furor. And as long as developers continue to tolerate unsatisfactory noise levels, and work longer hours to compensate for their negative effect on their productivity, organizations will continue to ignore their dissatisfaction.

---

[*] First published 11 Sep 2005 at http://www.hacknot.info/hacknot/action/showEntry?eid=78
[1] *Death to the Cubicle!*, Linda Tischler, FastCompany, June 2005
[2] *The Man Behind the Cubicle*, Yvonne Abraham, Metropolis, November 1998
[3] *"Resolve" product literature*, Herman Miller
[4] *Agile Software Development*, Alistair Cockburn, Addison Wesley, 2002
[5] *Human Performance Lecture*, Dr Nick Neave, Northumbria University
[6] *Collaborative Knowledge Work Environments*, J. Heerwagen, K.Kampschroer, K. Powell and V. Loftness
[7] *Peopleware*, T. DeMarco and T. Lister, Dorset House, 1987

Interview with the Sociopath*

Recently I have had the misfortune to be playing the interview circuit again; parading from one interrogation to the next like some prisoner of technical war. The experience has been both frustrating and humiliating – and unpleasant reminder of how appallingly most technical interviews are conducted.

So ignorant is the conduct of many interviewers, one could be forgiven for thinking they have undertaken the interview process with the deliberate intent of minimizing the chances of finding the right person for the job, and maximizing the opportunity for their own ego gratification. Such behavior is a common feature of the sociopathic personality.

Based on my recent interview experiences, I've assembled below a list of the techniques commonly practiced by the sociopathic interviewer.

Put No Effort Into The Position Description

The best way to ensure you don't accidentally get the right person for the job is to have no idea who you're looking for and what role they will be fulfilling in your organization. A meager and perfunctory PD (position description) helps to convey that "don't care" attitude right from the start of the hiring process. If you're working through a recruiting service, simply tell the recruiter that you don't have time to write out a decent PD. Rattle off a few buzzwords and acronyms and leave them to patch something together themselves.

If you are somehow compelled to write a PD, fill it out with the usual platitudes about "excellent communication skills", "ability to work well in a team", "delivering high quality code" ... and other such nonsense that 90% of programmer PDs include and which nobody can effectively appraise in an interview situation.

Conduct Phone Interviews With A Poor Quality Speakerphone

Phone interviews provide an excellent opportunity to explore the aural aspects of discourtesy. Always use a low quality speakerphone; even if you are the sole interviewer. Make the call from the largest, echo-filled room that you have access to, and sit a long way from the speakerphone. If there is more than one interviewer, make sure you constantly interrupt and talk over each other, making it impossible for the candidate to distinguish who they are currently talking to. The frustration of the constant struggle to understand and be understood will eventually wear down even the most ardent of candidates, often with comic effect.

Be Poorly Organized

Some candidates have the audacity to view the organization of an interview as being representative of the organizational capabilities of your company as a whole. They reason that finding someone to fill a role is effectively a mini-project in itself, and if you can't schedule and coordinate even a minor project like that, how could you manage a larger and more complex undertaking like a software project? These people are clearly thinking too hard and too critically. They are exactly the ones that you want to turn off. Therefore you should make every effort to have the interviewing process reflect the abysmal state of project management in your company as closely as possible.

Demonstrate your inability to estimate and track tasks by scheduling candidates' interviews too close together, booting one candidate out the door just as the next is about to give up hope that their own interview will ever commence. Having started the interview late, make it clear from the outset that you don't have much time to devote to each individual so you will have to rush. This will demonstrate your tendency to meet deadlines by making heroic efforts rather than rational adjustments of scope.

Then reveal that you have no questions prepared for the candidate. Just “um” and “ah” your way through a random series of queries that reveal no overall structure or intent, thereby conveying your inability to structure a work effort appropriately.

Focus On Technical Arcana

Technical interviews are a sociopath's utopia, for they provide you with infinite opportunity to humiliate a candidate while engendering feelings of supreme inadequacy. Even if a candidate has been using a particular technology for many years, chances are that they have only dealt with the most commonly used 80% or so of that technology's features. Therefore your questions relating to that technology should target the seldom encountered 20% at the periphery. Identify those aspects of the technology so infrequently used that most developers have either never been called upon to use them, or if they have, have not done so sufficiently to internalize the finer points of its operation. Drill the candidate mercilessly on these obscure and largely irrelevant details. When they fail to provide the correct answers, assume a facial expression that betrays your amazement that they have managed to survive in the industry without having immediate recall on every aspect of the technology they deal with.

Hire A List Of Products And Acronyms, Not A Person

The topic of "business value" should be avoided at all costs. Do not ask about the candidates' contributions to the businesses they have worked in, as this implies that all that boring business stuff is actually of concern to you. The sort of person you want is one who is solely focused upon decorating their CV with the latest buzzwords, and playing around with whatever "cool" technologies that vendors have most recently grunted out. You'll get such a person by ignoring the business aspect of software development, and assessing candidates solely on the amount of technical trivia they know. Clearly, those who take a "technology first" approach are motivated more by self-interest than professional responsibility, and are more likely to be suitable company for the sociopathic interviewer.

Pose Unsolvable Problems

A favorite ploy of sociopathic interviewers everywhere is to ask questions that have no concrete answer. The standard defense of this technique is the claim that it verifies the candidates' ability to take a logical approach to problem solving. Of course, there is no empirical evidence correlating the ability to solve logic puzzles with the ability to develop software - but no matter.

The real reason for asking questions that permit no solution is to watch the candidate squirm "on the hook", and to experience that feeling of smug self-satisfaction that you get when you finally acknowledge that there is no solution to the problem – it's just an exercise.

Such questions include:

- "How would you count the number of gas stations in the US?"
- "How would you measure the number of liters of water in Sydney Harbor?"
- "How would you move Mount Fuji?"

... which are all variants on the classic quandary "How long is a piece of string?" and equally deserving of serious consideration.

Ask About MVC

For some reason, it has become accepted in technical circles that all programming interviews must contain a question about the Model-View-Controller pattern. Every candidate expects it, every interviewer asks it – and there's no good reason for you to challenge the tradition. At least it chews up some interview time and spares you having to think of your own questions.

Ask General Questions But Expect A Specific Answer

This technique is the staple of anti-social interviewers everywhere. It's particularly handy if you want to devote no cognitive energy whatsoever to the proceedings. Ask a question that is general enough to permit multiple answers, but badger the candidate until they provide the specific answer that you have in mind. Thus a technical query turns into a guessing game, which is great fun for everyone – providing you're not the one doing the guessing.

Take Every Opportunity To Demonstrate How Clever You Are

For the sociopath, the interview is mainly about them and only peripherally about the candidate. They view an interview as an opportunity to demonstrate their natural intellectual and technical superiority. That they control the questions and have had time to research the answers doesn't hurt either.

You should make frequent, derogatory references to the quality of the candidates you have previously interviewed, the implication being that the current candidate can expect to be discussed in similarly negative terms once they are absent.

Don't hesitate to mock the candidate if they answer a question incorrectly. If it looks like they are about to provide a correct answer, interrupt them and change or augment the original question with additional complexities, creating a moving target that they will eventually abandon hope of ever hitting.

A technique that will certainly annoy the candidate (and people react in so much more interesting ways once they're angry, don't they?) is to deliberately misinterpret the candidates answer, exaggerate or distort it, then throw it back to them as a challenge i.e. create a straw man from their answer. Here is an example from one of my recent interviews:

Interviewer: *Have you participated in code reviews before?*

Ed: *Yes. I've reviewed other team member's code on many occasions.*

Interviewer: *So you don't trust your colleagues, then?*

An attitude of willful antagonism will enable you to goad even the most dispassionate of candidates into an angry (and entertaining) response.

Set COMP101 Programming Problems

Companies intent upon creating the impression that they really care about the quality of their people will give potential candidates a hokey COMP101-level programming problem to solve prior to granting them an audience. The solution provided is then dissected carefully and assessed according to criteria that the candidate was not made aware of at the time the assignment was set. Ridiculous extrapolations and inferences about the author's general programming ability are then made based upon the given code sample.

The beauty of this technique is that because the problem has been offered context-free, the candidate has no idea what design forces should influence their solution. They don't know what importance to assign to non-functional criteria such as performance, extensibility, genericity and memory consumption. The weight of these factors might significantly influence the form of the solution. By withholding them, and because these factors are often in conflict with each other, it is impossible for the candidate to submit a solution that is correct. Simply change the criteria for evaluation to the opposite of whatever qualities their solution actually contains.

For example, if their solution is readily extensible, claim that it is too complex. If they have favored clarity over efficiency, criticize their solution for its verbosity and memory footprint. If they have provided you only with code, select documentation-level and handover-readiness as the criteria-du-jour – question the absence of release notes.

Treat Senior Candidates The Same As Junior Candidates

Those who have been in the industry for a few decades will probably arrive at the interview expecting you to draw upon their extensive experience as a source of examples of problems you have solved, applications you have implemented and difficulties you have overcome. A sociopathic interviewer should demonstrate their contempt for the candidates' life's work by completing ignoring their work history. Make it clear that you don't care about the past by treating even the most senior of candidates like a fresh-faced rookie, demonstrating an appropriately condescending and patronizing attitude. After all, even the most worldly-wise candidate appears naive when put alongside your own towering genius.

The most effective means of convey your disdain for the candidate that I have witnessed is to ask them to take an IQ test, thereby implying that it is not their professional qualifications which are in doubt, but their native intelligence.

Make The Interview Process Long And Arduous

There is a lot of folk wisdom surrounding the hiring process. One common misperception is that the more arduous the interview process (i.e. the more rounds it contains, the greater the size of the interview panel etc.) then the more worth the position actually has. In other words, the harder the journey the better the destination must be. Clearly, the logic is flawed – it is quite possible for a long and demanding journey to conclude in a cesspit.

In an organizational context, a protracted interview process may simply indicate that the company is disorganized, indecisive and have failed to gather the information they needed in an efficient manner. But the myth persists, so you can exploit it to maximum effect, creating ever greater hoops for the candidate to jump through, on the pretext that you are being thorough or somehow testing their commitment. Be careful not to let on that you are really only demonstrating your own ineptitude and disrespect for the candidate's time.

Don't Hire Too Smart

One of the biggest hiring mistakes you can make is to hire someone who is better than you, and whose subsequent performance makes you look bad by comparison. As soon as you've formed an impression of the candidate's ability, adjust your interview technique accordingly. If the candidate is too good, step up the difficulty and obscurity of the questions you ask until you reach the point where they are struggling, and thereby creating a bad impression with any other interviewers present. If you sense the candidate is just good enough to do the job but not so good that they could do your job, then ease up on the questions and let them shine.

Remember that there may also be some career advantage in simply not filling the position at all; concluding that you simply couldn't find a suitable candidate. You may be able to emphasize how lucky your company was to have hired the last decent software developer out there – you.

Conclusion

The senior ranks of the software development community seem to attract more than it's fair share of sociopaths. Such people undertake the interview process with the same intent as they approach all activities – to create advantage for themselves. Whether you are amongst the self-adoring community of psychopaths, or just anti-social with psychopathic ambitions, the technical interview is a professional construct designed with your particular needs in mind. Using the techniques described above, interviews can be both a means of self-gratification and a fulcrum for leveraging your own career advantage.

---

[*] First published 24 Nov 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=70

The Art of Flame War*

The word "argument" has negative connotations for many people. It is associated with heated exchanges and passionate disagreement. But your experience of argument need not be so negative. Consider that the word 'argument' also means 'a line of reasoning'. By approaching a verbal or electronic discussion, even a hostile one, with this definition in mind, you can learn to separate the logical content of the exchange from its emotional content and thereby deal with each more effectively. You may even find the process of so doing an agreeable one.

The following are a few tips and techniques that I've learnt in the course of a great many arguments, flame wars and other "vigorous discussions" that may help you argue more purposefully, and thereby come to view argument as a stimulating activity to be relished, rather than an ordeal to be avoided.

You Can Be Right, But You Can't Win

At the end of a formal debate, one or more adjudicators decides which team are the victors. If only it were that clean cut in real life. A good portion of the time, arguments arise spontaneously, continue in a haphazard manner and then fizzle out without any clear resolution or outcome. When you cannot force your opponent to concede their losses or acknowledge your victories, it becomes impossible to keep score. Therefore you should not enter any dispute, particularly an online one, with visions of your ultimate rhetorical triumph, in which you lord your argumentative superiority over your opponent, who shirks away, cap in hand and ego in tatters. It's not going to happen.

So why engage in argument at all, if you can never win? Here are a few possible motivations:

- To hone your rhetorical and logical skills i.e. your attitude will be more playful than combative
- To get something off your chest
- To gratify your ego
- To restore the balance of opinion
- To humiliate your opponent

- To defend your own beliefs against a real or perceived attack
- To learn about your opponent
- To learn about yourself
- To explore the subject matter
- To protect your reputation against a real or perceived slight

Remain As Dispassionate As Possible

This is at once the most difficult and the most valuable aspect of arguing effectively. Strong emotion can cloud your thinking and inhibit your ability to reason objectively and thoroughly. Anger is what turns a discussion into an argument and then into a flame war. Responses you give while angry are likely to be poorly considered, so it is invaluable to have techniques at your disposal to moderate that anger so that you can argue at your best and even begin to enjoy the dispute. Here are a few techniques that might be useful:

- When you're not arguing in real-time (e.g. via email or discussion forums), print out the email or message that you've found inflammatory. Read it somewhere away from the computer and plan how you will respond. Delay making your actual response as long as possible.

- When arguing in person, make a deliberate effort to slow down the pace of the discussion and lower its volume. If you're uncomfortable with the silence created, adopt a thoughtful expression and pretend to be considering your reply carefully. Use the time created to take a few deep breaths and calm down.

- Adopt a different mental posture towards the email or message. Pretend that the message is for someone else. This helps to de-personalize the argument and put it at a distance.

Realizing that your opponent is as susceptible to emotion as you are, you may choose to use this to your advantage. Here we venture out of the realm of the logical and into the rhetorical. If you can identify your opponent's "hot buttons," then you may be able to goad them into making an unconsidered response. Once made, the response cannot be retracted and you may be able to play that advantage for the remainder of the argument. When being inflammatory or provocative, be careful not to overdo it. Lest you appear vitriolic or juvenile, make your barbs short and well targeted. Ensure that they are offered as parenthetical asides rather than as a basis for argument.

Perhaps the most effective means of disarming your opponent's insults is with wit, as demonstrated by the following exchange between Winston Churchill and Lady Asbury:

Lady Asbury:	*Mr. Churchill, if you were my husband, I would put poison in your wine.*
Churchill:	*Madam, if you were my wife, I would drink it.*

Be Familiar With The Basic Logical Fallacies

Those not skilled in argument are often prone to employing logical fallacies and being unaware that they are doing so. It is vital that you be able to recognize at least the basic logical fallacies so that you don't end up trying to attack an insensible argument, or formulating one yourself. Common logical fallacies include:

- *Straw Man Argument* – Your opponent restates your argument inaccurately and in a weaker form, then refutes the weaker argument as if it were your own.
- *Argumentum Ad Hominem* – Ad hominem means 'to the man'. Your opponent attacks you rather than your argument. If you choose to insult your opponent in order to provoke an emotional reaction, be sure that your insults are not used as part of your argument, otherwise you will be guilty of argumentum ad hominem yourself.
- *Appeal to Popularity* – The suggestion that because something is popular it must be good, or because something is widely believed it must be true.
- *Hasty Generalization* – Making an unjustified generalization from too little evidence or only a few examples.
- *Appeal to Ignorance* – Claiming that something is true because there is no evidence that it is false.
- *Appeal to Authority* – Claiming that something is true because someone important says that it is.

Seek Precision

It's easy to end up arguing at cross-purposes with someone simply because you each have different definitions in mind for component terms of the subject being debated. So a good starting point when engaging in debate is to first ensure that you and your opponent have precisely the same understanding of the topic being argued. Remarkably often, the act of precisely defining the topic will serve to circumvent any subsequent argument, as it becomes clear that the warring parties do not have conflicting positions on a given subject, but instead are talking about different subjects entirely.

Ask Pointed Questions

There are several reasons why you might choose to ask your opponent questions:

- To seek clarification on a point that they have made
- In the hope that some of the information volunteered will be faulty, thereby providing you with fuel for rebuttal.
- To save effort on your part. It often takes less effort to ask a question than answer it. In a protracted exchange, this economy of effort can be important. It also gives you time to think about your next move.
- Because you know the answer. A powerful rhetorical technique is to ask a series of questions that leads your opponent, by degrees, to the realization that their answer is in contradiction with statements they have previously made.

For example, suppose you are arguing the merits of free software with one of Richard Stallman's disciples. You might question the Free Software Advocate (FSA) to tease out the inconsistencies in their philosophy:

FSA: *All software should be "free", as in "freedom"*

You: *How do define "free", exactly?*

FSA: *"Free" means that you can do with it whatever you want.*

You: *With no restrictions at all?*

FSA: *Yes - you have absolute freedom to do with it whatever you please. Anything else is an attempt to take away your* *freedom.*

You:	*Then I would be free to make it non-free if I wanted to?*
FSA:	*Ummm ... I guess so.*
You:	*But wouldn't that contradict your original statement that "all software should be free"?*

If the last response from the FSA had been different, the argument might have headed in a different direction:

You:	*Then I would be free to make it non-free if I wanted to?*
FSA:	*No - that's the exception. You can't inhibit the freedom of others.*
You:	*But doesn't that mean that I'm not really free? Specifically, I'm not free to inhibit the freedom of others?*
FSA:	*Sure, but you have to draw the line when it comes to fundamental liberties.*
You:	*And what basis do you have for claiming that free use of software is a fundamental liberty?*

... and so the FSA is led to an awareness of the circular reasoning they are employing.

Don't Claim More Than You Have To

A common error is to extend the claims you're making to a broader scope than is really necessary to make your point. In doing so, you extend the logical territory that you have to defend and permit counter-argument on a broader front. This is one of the primary benefits of maintaining a skeptical attitude. Skeptics assume as little as possible, and therefore have less to defend than True Believers who are prone to making broad assumptions and sweeping generalizations.

Suppose you're arguing about the quality of open source software versus proprietary software. An open source zealot may make a broad claim such as "Open source software is always of higher quality than proprietary software". A universal qualifier such as "always" makes their claim easy to disprove – all that is required is a single counter-example. A more cautious open source enthusiast might claim "Open source software is *usually* of higher quality than proprietary software", which is a narrower claim than the one made by the zealot, but one still requiring evidential support. A skeptic might ask "How do you define quality?"

Claims can be accidentally over-extended by provision of a flawed example of the general point you're making. Your opponent counters the particular example you've provided and then assumes victory over the general claim it was supposed to be illustrating. Before choosing to illustrate your general claim with a specific example, be very sure the example is a true instance of your general case. It may be more prudent to leave out your example all together.

Seek Evidence

It's easy to make bold claims and impressive assertions; it's not so easy to back them up with proof. A common problem in argument is the failure to identify which party carries the burden of proof, and to what extent that burden exists. The general rule is this: He who makes the claim carries the burden of proving it. If you claim "Linux is more reliable than Windows", then it is your responsibility to not only specify your definition of "more reliable" but to provide evidence that supports your claim. Your claim is not "provisionally true" until someone can prove you wrong; and neither is it false. It's truth or otherwise is simply unknown.

This is an area of common misunderstanding amongst those with pseudo-scientific beliefs. For instance, UFO believers will look at a history of UFO sightings for some region and note that although 99% have been attributed to aircraft, weather balloons and such, 1% of them are still unexplained. They delight in this 1% figure as if it were vindication of their beliefs. But 1% being "unknown" does not equate to "1% being alien beings in spaceships". It might also mean that the 1% of reports were simply too vague or incomplete to permit any kind of conclusion being reached. Those claiming by implication that the 1% represent alien beings carry the burden of proving that with evidence.

But always remain aware of the context in which claims are made. Different contexts bring with them different levels of formality, and consequently different evidentiary standards. If your friend remarks "Boy it's hot outside", it's obviously not appropriate to insist upon meteorological data to back up their claim. But if an environmental activist claims "average daytime temperature world-wide has risen an average of 0.5 degrees in the last century" then the first thing you'll be wanting to know is where the data came from that supports that claim.

When Your Opponent Is Irrational

Finally, there is a delicate ethical issue to consider when arguing. Every so often you find yourself locking horns with someone who appears to have a fairly shaky grip on reality. I'm not referring to simple eccentricity or religious fervor, but psychiatric illness. For examples, you can refer to some of the emails received by the James Randi Educational Foundation[1] (JREF) in response to their million dollar challenge. James Randi is a well known skeptic and magician. Since 1994, the JREF has offered a prize of one million dollars to anyone able to demonstrate paranormal or supernatural abilities or phenomena under controlled observational conditions. To date, no one has successfully claimed that prize. But some of the applications[2] they receive suggest that the respondent is unwell, perhaps delusional. If you should find yourself in online discussion with someone whom you suspect is unencumbered by the restrictions of rational thought, then perhaps the best you can do is exit the discussion immediately. To continue is to risk antagonizing someone who may be genuinely dangerous. This is one of the prime reasons for conducting online arguments anonymously, where possible.

Knowing When To Quit

There comes a point when you want to exit an argument. Perhaps you've grown bored with it; perhaps it has become clear that your opponent's views are so heavily entrenched that progress is impossible; perhaps your opponent is offering only insults without any logical content. Here are a few ways of bringing the argument to a definite conclusion, rather than just letting it peter out:

- Simply walk away. For online arguments, refuse to respond.
- Insist that any topics covered thus far be resolved before the argument continues. This prevents your opponent switching subjects and responding to your rebuttals by simply making a new batch of assertions.
- Ask your opponent what they hope to gain by continuing the argument. To what end are they arguing.

Reconstruct Your Opponent's Argument

Argument reconstruction is the process of analysis the verbal or written form of an argument and identifying the premises (both explicit and implied) and the conclusion/s it contains. To effectively rebut your opponent's arguments you need to know exactly what they are claiming, and upon what basis they are claiming it. For each premise you identify, consider whether the premise is true or false. If you think one or more of them is false, call attention to each of them and ask your opponent to justify them with evidence. If the conclusions don't follow logically from the premises, call attention to the logical error. If the conclusion cannot be true without one or more unstated premises also being true, then call your opponent's attention to their reliance upon implicit premises and, where those premises are in doubt, insist that evidence be provided in support of them.

---

[*] First published 13 Mar 2005 at http://www.hacknot.info/hacknot/action/showEntry?eid=72

[1] http://www.randi.org/

[2] http://forums.randi.org/forumdisplay.php?f=43

Testers: Are They Vegetable or Mineral?*

There are real advantages to having a group of people, separate from developers, whose job is solely to find fault with your work. They have an emotional and cognitive distance from the product that a developer can never fully imitate. Testing is a task requiring patience, attention to detail and a fairly devious mindset. Sometimes managers make the mistake of regarding testing as a second class activity, suitable to be performed by less skilled or more junior staff members. Such misimpressions are a disservice to the project and the testing community.

But a common byproduct of having a distinct testing team is the development of an adversarial dynamic between testers and developers. I can understand completely how easily this situation occurs. I recently had the misfortune to work with a testing team whose methods left myself and other developers ready to kill them.

Below, I have listed the main work habits this team engaged in, that made them so difficult to work with. I hope that these items may serve as a brief catalog of bug reporting "anti-patterns" that testers can use as a checklist to make sure they are not accidentally annoying the developers they work with, and that developers can use to identify sources of friction between themselves and their testing team.

Abbreviating Instructions For Reproducing The Bug

Problem: Some testers believe that they can save themselves some time by describing the circumstances under which the bug appears in the briefest terms possible. Often the bug report degrades into a contracted narrative that only specifies the milestones in the series of actions necessary to reproduce the bug. Being unfamiliar with the application's internal structure, a tester can not know which of the series of actions they have followed is most significant when diagnosing the underlying fault. By neglecting actions they consider unimportant, there is a significant risk they are omitting important information.

Solution: The best way to avoid this is to simply enumerate *all* the actions that are necessary to reproduce the buggy behavior, starting with the launch of the application. Put the first step in a bug reporting template to remind testers to do this e.g. "1) Launch the application. 2) *your text here*"

Not Identifying The Erroneous Behavior

Problem: The description in the bug report ends in a simple statement of application state without identifying what aspect of that state is actually in error. For example, the bug report concludes "The Properties dialog appears", but the tester fails to add "... and the property controls are enabled, even though the selection is read-only".

Solution: Put the heading "Erroneous behavior:" or "Actual behavior:" in your bug report template, to remind the tester to include that information.

Not Identifying The Expected Behavior

Problem: Even when the bug report contains a description of the erroneous behavior, testers sometimes forget to explain what the expected (correct) behavior is. For example, the bug report concludes "The file saves silently", but the tester fails to add "... but there is no visual indication that the application is busy performing the save. The cursor should change to an hour glass and a modal progress dialog should appear.

Solution: Put the heading "Expected behavior: " in your bug report template, to remind the tester to include that information.

Not Justifying The Expected Behavior

Problem: It is not always clear why a tester has decided that a particular behavior is buggy. The bug report may simply claim "X should happen" without making it clear why X is the correct behavior. A reference to a requirement specification is an appropriate justification. If that requirement is for adherence to an externally specified standard, then a reference to the relevant portion of that standard is appropriate.

Solution: Put the heading "Requirement reference:" in your bug report template, to remind the tester to include that information.

Re-Opening Old Bug Reports For New Bugs With Similar Symptoms

Problem: A bug report is marked as FIXED and everyone thinks it is done with. But in the course of subsequent testing, a tester sees faulty behavior occurring that is very similar to that produced by the bug that was thought FIXED. Reasoning that the behavior is so similar that it must have the same underlying cause, the tester concludes that the bug previously marked FIXED has resurfaced. They REOPEN the FIXED bug report. This is problematic for the developer, because the re-opening of the bug implies that the original symptoms are re-occurring, not the similar symptoms that the tester is now observing. The tester has communicated to the developer their incorrect diagnosis of the fault, rather than simply reporting the faulty behavior they have observed.

Solution: Insist that testers refrain from reusing old bug reports unless the erroneous behavior they see is exactly the same as that described in the old bug report. Even then, there is some chance of confusing two separate bugs that just happen to produce identical observed behavior. If there is any doubt, create an entirely new bug report. The develop can always mark it as a duplicate of the old bug report and re-open the old bug report themselves, if investigation demonstrates that the new and old bugs have the same underlying cause.

See also "Diagnosing Instead of Reporting"

Testing An Old Version Of The Software

Problem:

Developer:	*It's fixed!*
Tester:	*It's NOT fixed!*
Developer:	*It's fixed! Here's a screen shot showing it fixed!*
Tester:	*I don't care about your screen shot. It's NOT fixed for me!*

This developer / tester exchange quickly escalates into justifiable homicide and arises far more often than it should. In a testing process which permits the version of the software being tested to change underfoot, the conflict often arises from a developer fixing a bug in a version yet to be released to the tester. Both developer and tester are correct in their assessment of the bug's status, with respect to the version of the software that is front of them.

Solution: Institute a process to enable version coordination between developers and testers. Label each new version with a unique number and make the version numbers currently being tested and developed readily available to all. Ensure someone has the responsibility to update this version number whenever a new version is released to the testers. When a bug report is declared FIXED, ensure developers include the version number in which the fix will appear.

Inventing Requirements Based Upon Personal Preference

Problem: Generally a set of requirements is not so complete as to explicitly specify program behavior in every possible circumstance. Quite aside from inevitable oversights by those assembling the requirements, some requirements are left to "common sense". A requirement such as "shall conform to Microsoft Windows User Interface Guidelines" is broad and may be difficult to interpret in any particular instance. Rather than interrogate the standard thoroughly, some testers will try and substitute their own version of "common sense" for the requirement, bringing with it their mistakes and misinterpretations. For instance, I received a UI bug report indicating that "a sub-menu should not appear if all menu items within it are disabled." The tester regarded this as "common sense". However, the UI standards explicitly dictated that such sub-menus should always appear, even when all of their menu items are disabled, so that the user could at least see the contents of the sub-menu and would know where to find a particular option when it did become available. Yet the bug report stated quite emphatically that the behavior "should" be different. The tester had fabricated the requirement, and decided to lend it authority by using the word "should", so as to imply the presence of such a requirement.

Solution: See "Not Justifying the Expected Behavior"

Omitting Screen Shots

Problem: Many bug tracking systems provide the facility to attach a file to a bug report, the way one might attach a file to an email. But testers frequently forget (or can't be bothered) making use of this facility. Particularly for GUI-related bugs, a screen shot showing the bug occurring, or illustrating a step in its reproduction, is an efficient way of capturing information.

Solution: Make sure testers are aware of the "attach" functionality in your bug tracking system and are encouraged to use it. Image attachments can also be a convenient way of proving to a disbelieving developer that a bug occurs, or to a tester that a bug has been fixed.

Using Vague Or Ambiguous Wording

Problem: In the text of the bug report, the tester employs terminology that is imprecise or ambiguous. For example: the tester refers to "this dialog" in the bug report, intending the word "dialog" to mean "an exchange between parties"; but the developer interprets "dialog" as referring to a secondary window in the interface. Another example: The tester describes a text field as being "enabled when it should be disabled", but really intended that the text field is "editable when it should be uneditable".

Solution: None – however a large, blunt object applied with extreme prejudice can at least have a cautionary effect.

Diagnosing Instead Of Reporting

Problem: Either through arrogance or a misguided attempt to be helpful, the tester describes what they believe is the underlying fault exposed by the bug, rather than simply reporting the observed behavior. For example, the tester examines a log file and deduces from the name of an exception appearing in a stack trace that the application is running out of memory. Having provided this insight, they omit the rest of the bug report, thinking that they have already provided the crucial information.

Solution: See "Solution" above.

Exaggerating The Priority Of A Bug

Problem: Some testers exhibit a tendency to elevate the priority of the bug reports they lodge later in the testing process. As testing proceeds and the identification of new bugs becomes harder and harder, it seems that the extra effort involved in their location is justified by raising their priority - by way of psychological compensation, I suppose. Developers find that bugs which would have been regarded *minor* in early testing are suddenly becoming *major* issues. This effect may also be attributable to increasing stress or approaching deadlines.

Solution: For each priority level your bug reporting system allows, provide a clear definition that can be referred to in order to resolve disputes over bug priority.

Justifying Partial Coverage With Appeals To Bad Assumptions

Problem: Rather than exhaustively test all possible combinations of inputs or circumstances, testers choose a limited subset of these for testing, reasoning that the chosen subset will be sufficient to exercise the underlying code. In effect, they are making assumptions about the code coverage that results from manipulating the application's interface in various ways.

Solution: Sometimes assumptions of this nature can legitimately be made. If there is insufficient time to perform exhaustive testing, then it is the developers who should be choosing the representative subset of operations to test, not the testers.

See "Diagnosing Instead of Reporting"

---

* First published 13 Oct 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=68

Corporate Pimps: Dealing With Technical Recruiters*

Anyone who has had any substantial dealings with technical recruiters invariably has a poor opinion of them. This is because the standard of practice in the recruiting industry is so low. To be a recruiter you don't need any formal qualification, or any particular experience.

Recruiting, as it is generally practiced, is little more than telemarketing. As with telemarketing, people are drawn to it because of the opportunity to make money without having to satisfy any particular educational requirements. A recruiter's commission is generally 15-20% of the candidate's first year's salary, which explains why recruiters are not generally altruistically motivated. They share the ethical and moral shortcomings of workers in other commission-based occupations such as used car salesmen, real estate agents and pimps.

In your interaction with recruiters, it pays to keep the following firmly in mind:

- The recruiter is first and foremost a salesman, so their prime objective is to make money. They do this by finding someone who satisfies their client's requirements for long enough to earn them a commission.
- You don't need the recruiter's good favor, you just need to convince them to pass your resume onto their client. Because recruiters are universally maligned, their clients have no more respect for their opinions than you do.
- The recruiter has no technical knowledge. The skills you've spent years acquiring are just empty keywords and acronyms to them.
- Never allow yourself to be talked into doing something you don't want to. Recruiters are good talkers, and know how to railroad the introverted techie into a particular course of action. They will speak quickly, loudly and with unwarranted familiarity in order to influence you into doing what they want.
- Above all, remember that it's your career you're dealing with. You are the only one who exercises any control over that, not the recruiter.

When I began speaking with recruiters again recently, I went in search of a guide to help me deal with them more effectively. Finding no such guide available, I decide to write one. The following presents some tips on dealing with that most useless of creatures, the IT recruiter.

Phone Calls

Tip: Don't Bother Leaving Voicemails

You will find that recruiters rarely return your voicemail messages. The perceived justification for this discourtesy is "I'm too busy," although the real reason is "Contacting you doesn't hold the immediate promise of financial reward". Therefore, don't bother to leave messages – keep calling until you can speak to them in person.

Tip: Be Cautious When Answering Certain Questions

Recruiters will try and gather more information than is necessary, in the hope of learning something that can be used to their advantage. Only discuss what is strictly relevant to the job in question. In particular, look out for the following questions:

Do You Have Any Other Opportunities In Hand?

Recruiters will often make a "friendly enquiry" about how your job hunting prospects are at the moment. This is not idle small talk. The recruiter is trying to gauge:

- How desperate you are i.e. how much leverage they have
- The number of opportunities out there for people with your skill set. At best, this enquiry could be called "market research."
- The names of companies that are currently hiring – so they can approach them.

It is of no advantage to you to provide any of this information to the recruiter, and it could weaken your bargaining position in future. A suitable response might be "I'd prefer not to discuss the status of my job search." Above all, never appear desperate – it will be a signal to the recruiter that they can get away with dramatically cutting your rate, thereby increasing their profit margin.

What Recruiter Did You Apply Through?

If you tell them you have already made application for the position through another recruiter, they may try and find out who that recruiter is, and what agency they work for. It's none of their business – tell them so. The same response as above will suffice.

Do You Know Anyone Else Who Might Be Interested In This Job?

Here, the recruiter is trying to get you to refer them to another candidate. Never do this, if you want to keep your friends. Once that information gets into the recruiter's hands, there is no telling what will happen to it. The only appropriate answer to the above question is "no." If you do know someone who is interested, still tell the recruiter "no", and then contact that person yourself so they can approach the recruiter at their leisure, if they so choose.

Who Did You Work For While You Were At Company X?

A common technique recruiters use to broaden their client base is to use candidates to get contacts within companies the candidate has worked for. For example:

Recruiter:	Did you work for *fictional-name* while you were at J-Corp?
You:	No – I've never heard of *fictional-name*. I reported to John Smith.

Now the recruiter has a contact name within J-Corp that they can use to get past the company switchboard (companies often have switchboard blocks on recruiters). They can ring J-Corp's switchboard, ask to speak to John Smith – without revealing that they are a recruiter – and be in a position to market their services directly to someone who is reasonably senior.

What Was Your Rate/Salary In Your Last Contract/Job?

The danger in quoting a contract rate is that the rate at which you actually work (assuming you're awarded the contract) is yet to be negotiated. If the recruiter can subsequently negotiate a higher rate with his client, he can keep that information to himself and absorb the surplus into his margin.

Tip: Learn A Few Rote Answers

All recruiters tend to ask the same questions. It may surprise you to know that recruiters often follow scripts – the same way that telemarketers follow scripts when cold calling potential customers. They may have worked with the script so long that they've now internalized it, or perhaps they've developed the script themselves, refining it over the course of hundreds of phone calls. The point is, the recruiter is far more rehearsed in asking questions than you are in providing answers. To level the playing field, you can prepare your own scripts by rehearsing answers to some commonly asked questions:

Why Did You Leave Your Last Job?

Some recruiters will ask this, as if they had the right to know and could put the info to any sensible use. Prepare a brief and suitably vague answer that suggests you bear no animosity towards your last employer, and that your performance wasn't questioned in any way. A tried and true comeback is "It was just time for a change" – which is impossible to refute or question further.

What Is Your Ideal Job?

Occasionally a recruiter asks this, just on the off chance that your ideal job is currently on their books. Not surprisingly, it never is. They're not really interested in your response, so much as that you have one and asking it makes it sound like they're displaying due diligence. Learn a brief and dismissive answer.

Tip: Determine The Purpose Of The Call Early In The Conversation

It's not uncommon to have recruiters contact you even though they don't actually have a suitable position to discuss with you – the operative word being "suitable." You may find that they have a position that is clearly unsuitable for you, but will try and use that position to establish contact with you, ask you to come and see them for a chat, and generally begin the recruiting process. These recruiters are desperate and are trying to match the few positions they have to whatever candidature they can dig up, no matter how inappropriate the match. Don't let them waste your time. If they're not prepared to put a job specification down on the table, walk away.

Tip: Protect Your Referees From Unnecessary Interruption

There's no need to put "references available upon request" on your resume – that is understood. Out of consideration for your referees, you should aim to minimize the number of occasions they are contacted. Therefore, never give away your references until there is a job offer on the table, for the following reasons:

- Some recruiters will use your referees as contact points for marketing their services.
- If the recruiter contacts your referees, there is no guarantee that their client will not also want to contact them. Then your referees end up getting hounded with phone calls.
- If the recruiter contacts your referees prior to a job offer being made, and the client does not decide to hire you, then your referees have been pestered for nothing.

Some recruiters will try to tell you that they can't even submit your resume to their client without references. This is nonsense, and certainly an attempt to collect your referees as contacts.

Tip: Be Suspicious Of Calls From Agents You've Never Heard Of

Once you have been circulating your resume for a while, and it has been entered in the résumé databases of enough agencies, you'll find that you start getting cold calls from agents that you've never heard of. What's happened in these cases is that the agent has done a keyword search on their agency's résumé database for a particular skill set, got back several dozen matches, and then placed a phone call to every person whose resume was a match. Your resume happens to be in the agencies database as a result of your previous contact with some *other* agent working at that agency.

If an unknown recruiter leaves you a message, if you do call them back, you can expect the following:

- The recruiter doesn't remember who you are.
- The recruiter doesn't remember what job description they rang you in relation to.
- Once they've worked out those two things, they search their database for your résumé.

- Then they read out their job's skill requirements and you have to respond "yes" or "no" to each … even though that info is on the screen in front of them.

For this reason I generally don't return calls from recruiters I've never heard of. I have better things to do than read out my résumé over the phone.

Tricks Of The Trade

Trick: Bait And Switch

This is an old salesman's scam that still finds application in the recruiting industry. The practice consists of luring in a candidate with an inviting (but inaccurate or incomplete) job description, and once the candidate is "hooked", revealing the true nature of the position. The hope is that the sense of positive expectation already created will make the candidate more receptive to the true job description.

Trick: Salary/Contract Rate Negotiation

Never forget that the recruiter is paid by the client company to find employees, and he who pays the bill gets the service. Perhaps this is the way recruiters self-justify their poor treatment of candidates. It is also significant when the recruiter is negotiating a salary/rate on your behalf – they are negotiating with the same party that pays their commission, so it is as well to have a good idea of what money you're worth and to set definite boundaries for the recruiter so that you don't get sold out. Recruiters will try and get you to lower your rate by claiming that their client has one or more alternatives of similar experience/ability as yourself, and they are willing to work at a lower rate. You can never tell whether your competitors are real or are phantoms created by the recruiter. Any enquiries you might make to determine the authenticity of these competitors will be foiled by the recruiter's claims of privileged information.

Trick: Vague Job Descriptions

At times, recruiters will publish deliberately vague job descriptions in the hope of garnering as wide a response as possible. Their motivation is in part to refresh their internal resume database, and in part to assess the amount of interest associated with particular skills sets (market research). There may be an actual job behind it all, or there may not.

Trick: Agent Interviews

The "agent interview" is one of the biggest conceits in the recruiting industry. A small percentage of recruiters will want to speak with you in person before putting your résumé forward to their client. Some will even claim that they are required by company policy to do so. The ostensible purpose of these chats is for the recruiter to get a better idea of who you are, thereby enabling them to present your strengths more effectively to their client. If you were wondering exactly what a recruiter will learn about you in a 20 minute chat that they can't gather over the phone, then you wouldn't be the first. The real purpose of agent interviews are:

- For the recruiter to see how attractive you are. Statistically, good-looking candidates are more likely to interview successfully. If the recruiter has a choice of candidates to put forward, they are better off choosing the more attractive ones. Of course, discrimination based on appearance is illegal, so you'll never hear any public admission that this sort of assessment occurs.
- To increase your degree of investment in the agent and the job. Once you've gone to the effort of meeting with a recruiter, you will have a natural tendency in future to act in a way that retrospectively justifies having made that investment. In future you are more likely to favor that agent, and to be more kindly disposed towards positions put forward by that agent. If this sort of psychological manipulation strikes you as being beyond the average recruiter's capability, remember that most recruiters have at least an intuitive grasp of sales techniques. Exploiting your need to appear consistent with previous actions is a common technique employed by salesmen. The door-to-door salesman who offers a free demonstration of his product knows that the hidden expense is the cost of your time, which is only justified if you later make a purchase. The car salesman who lets you take a vehicle for a test drive is relying upon the same principle.
- To establish a power dynamic. It is significant that you go to the recruiter, and not the other way around. This suggests that the recruiter is in control, as they would like to believe, and as they would like you to believe.

Trick: X-Rayers And Phone Lists

Recruiters will go to extraordinary lengths to get leads to clients and candidates. There are a number of software packages available, called web site “x-rayers” or “flippers”, designed to automatically probe corporate websites for names and phone numbers. Lurking on Usenet groups is another way of getting relevant email addresses. Looking to fill a Java job? A few weeks lurking on `comp.lang.java` enables the recruiter to identify the technically savvy and geographically appropriate posters. I suspect the vast majority of recruiters are not technically savvy enough to use these sorts of techniques. However, that such possibilities exist does illustrate why it’s worthwhile being very careful with how much information you give away.

Trick: Wooden Ducks

Particularly unscrupulous recruiters will submit candidates to their client to act as placeholders – for the purposes of making another candidate appear good by comparison. It’s going to be difficult to determine when you are being used as a wooden duck because you have no knowledge of the other candidates your recruiter is putting forward. Tell tale signs may be:

- The recruiter is pushing hard for you to attend an interview, even though they have previously expressed doubts about your chances against other candidates.
- The recruiter makes no effort to coach you about the interview, what to expect or how to prepare.
- The recruiter has hinted that you may be competing against internal candidates i.e. candidates already employed by the client.
- The recruiter has made statements such as “not getting your hopes up” or similar, indicating they are anticipating failure.

* First published 12 Jul 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=1

Developers are from Mars, Programmers are from Venus*

Many of us use the terms "programmer" and "developer" interchangeably. When someone asks me what I do for a living I tend to describe my vocation as "computer programmer" rather than "software developer", because the former seems to be understood more readily by those unfamiliar with IT. Even when writing pieces for this site, I tend to swap back and forth between the two terms, to try and avoid sounding repetitive. But in truth, there is a world of difference between a computer programmer and a software developer.

The term "programmer" has historically referred to a menial, manual input task conducted by an unskilled worker. Predecessors of the computer, such as the Hollerith machine, would be fed encoded instructions by operators called "programmers". Early electro-mechanical, valve and relay-based computers were huge and expensive machines, operated within an institutional environment whose hierarchical division of labor involved, at the lowest level, a "button pusher" whose task was to laboriously program the device according to instructions developed by those higher up the technical ladder. So the programmer role is traditionally concerned only with the input of data in machine-compatible form, and not with the relevance or adequacy of those instructions when executed.

A modern programmer loves cutting code – and *only* cutting code. They delight in code the way a writer delights in text. Programmers see their sole function in an organization as being the production of code, and view any task that doesn't involve having their hands on the keyboard as an unwanted distraction.

Developers like to code as well, but they see it as being only a *part* of their job function. They focus more on delivering value than delivering program text, and know that they can't create value without having an awareness of the business context into which they will deploy their application, and the organizational factors that impact upon its success once delivered.

More specifically ...

Developers Know Something Of The Domain And The Business

Programmers like to stay as ignorant as possible of the business within which they work. They consider the problem domain to be the realm of the non-technical, and neither their problem or concern. You'll hear programmers express their indifference to the business within which they operate - they don't care if it's finance, health or telecommunications. For them, the domain is just an excuse to exercise a set of programming technologies.

Developers view the business domain as their "second job." They work to develop a solid understanding of those aspects of it that impact upon their software, then use that knowledge to determine what the real business problems are that the application is meant to be solving. They make an effort to get inside the heads of their user base – to see the software as the users will see it. This perspective enables them to anticipate requirements that may not have occurred to the users, and to discover opportunities to add business value that the users may have been unaware was technically possible.

Developers Care About Maintenance Burden

Programmers crave new technologies the way children crave sweets. It's a hunger that can never be satiated. They are forever flitting from one programming language, framework, library or IDE to the next; forever gushing enthusiastically about the latest silver bullet to have been grunted out by some vendor or open source enthusiast, and garnished with naive praise and marketing hype. They won't hesitate to incorporate the newest technology into critical parts of their current project, for no reason other than that it is "cool", and all the other kids are doing it. They will be so intent on getting this new technology working, and overcoming the inevitable troubles that immature technologies bring, that there will be no time to spare for documentation of their effort. Which is exactly how they like it – because documentation is, they believe, of no use to them. Sure, it might be useful to future generations of programmers, but who cares about them?

Developers have a much more cautious approach to new technology. They know that a new technology is inevitably hyped through the roof by those with a vested interest in its success, but that the reality of the technology's performance in the field often falls short of the spectacular claims made by proponents. They know that a technology that is new is also unproven, and that its weaknesses and shortcomings are neither well known or publicized. They know that part of the reason it takes time for the negative experiences with technologies to become apparent is that many developers will be hesitant to say something critical amongst that first flush of community enthusiasm, for fear that they will be shouted down by the newly-converted zealots, or dismissed as laggards who have fallen behind the curve. So developers know to stand back and wait for the hype to die down, and for cooler heads to prevail. Developers also know the organizational chaos that can result from too many changes in technical direction. A company can quickly accumulate a series of legacy applications, each written in a host of once-popular technologies, that few (if any) currently on staff possess the skills to maintain and extend. Those that first championed those technologies and forced them into production may have long since moved onto other enthusiasms, perhaps other organizations, leaving behind the byproduct of their fleeting infatuation as a maintenance burden for the organization and future staff to bare.

Developers Know That Work Methods Are More Important Than Technical Chops

Programmers often focus so intently upon the technologies they use that they come to believe that technology is the dominant factor influencing the ultimate success or otherwise of their projects. The mind set becomes one of constantly looking over the horizon for the next thing that might solve their software development woes. The expectation becomes "Everything will be better once we switch to technology X."

Developers know that this "grass is greener" effect is a falsehood – one often promulgated by vendors, marketers and technology evangelists in their quest to sell a product. The dominant factors influencing the quality of your application, and ultimately its success or otherwise, are the quality of the people doing the development and the work methods that they follow. In most cases, technology choice is almost incidental (the one possible exception being where there is a generational, revolutionary change in technology, such as the transition from low level to high level programming languages). Therefore developers frequently posses an interest in QA and software engineering techniques that their programmer counterparts do not.

Programmers Try To Solve Every Problem By Coding

It is characteristic of the programmer mentality that every problem they encounter is perceived as an opportunity to write more code. A typical manifestation is the presence of a "tools guy" on a development team. This is the guy who is continually writing new scripts and utilities to facilitate the development process, even if the process he is automating is only performed once in a blue moon, meaning that there is more effort expended in writing the tool than the resulting automation will ever save.

Developers know that coding effort is best reserved for the application itself. After all, this is what you are being paid to produce. They know that tool development is only useful to a point, after which it becomes just a self-indulgent distraction from the task at hand. Typically, a retreat sought by those with a love of "plumbing" and infrastructure-level development. Developers know that there are many development tasks that it is simply not worth automating and, where possible, will buy their development tools rather than roll their own, as this is the most time- and cost-efficient way of meeting their needs.

Developers Seek Repeatability, Programmers Like One-Off Heroics

If development were an Aesop's fable, then programmers would be the hares, and developers the tortoises. Programmers, prone to an over-confidence resulting from excessive faith in technology's ability to save the day, will find themselves facing impending deadlines with work still to go that was meant to be made "easy" by that technology, but was unexpectedly time-consuming. Not surprisingly, the technology doesn't ameliorate the impact of too little forethought and planning. These last-minute saves, and the concentrated effort they require, are later interpreted as evidence of commitment and conviction, rewarded as such, and thereby perpetuated.

Developers are very aware that there are no silver bullets, be they methodological or technological. Rather than pinning their hopes on new methods or tools, they settle down to a period of detailed analysis and planning, during which they do their best to anticipate the road ahead and the sorts of obstacles they will encounter. They only proceed when they feel that they can do so without entertaining too much risk of making faulty assumptions, and having to later throw work away.

Programmers Like Complexity, Developers Favor Simplicity

It's not uncommon for programmers to deliberately over-engineer the solutions they produce, simply because they enjoy having a more complex problem to solve. They may introduce requirements that are actually quite unnecessary, but which give them the opportunity to employ some technology that they have been itching to play with. Their users will have to bear this extra complexity in their every interaction with the system; maintenance programmers will have to wade through it in every fix and patch; the company will have to finance the extensions to the project schedule necessary to support the additional implementation effort; but the programmers care about none of this – as long as they get to play with a shiny new tech toy.

Developers continually seek the simplest possible resolution to all the design forces impinging on their project, regardless of how cool or trendy the technology path it takes them down. If the project's best interests are served by implementing in Visual Basic, then VB is what you use, even though VB isn't cool and may not be something you really want to see on your CV. If the problem doesn't demand a distributed solution, with all the scalability that such an architecture provides, then you don't foist a distributed architecture upon the project just so you can get some experience with the technologies involved, or just because it is possible to fabricate some specious "what if" scenario to justify its usage, even though this scenario is never likely to occur in a real business context.

Developers Care About Users

Programmers often view their user base with disdain or even outright contempt, as if they are the ignorant hordes to whose low technical literacy they must pander. They refer to them as "lusers", and laugh at their relative inexperience with computing technology. Their attitude is one of "What a shame we have to waste our elite programming skills solving your petty problems" and "You'll take whatever I give you and be thankful for it." Programmers delight in throwing technical jargon at the user base, knowing that it won't be understood, because it enables them to feel superior. They are quick to brush off the user's requests for help or additional functionality, justifying their laziness by appealing to "technical reasons" that are too involved to go into.

Developers don't consider users beneath them, but recognize and respect that they just serve the organization in a different capacity. Their contribution is no less important for that. When speaking with users, they try to eliminate unnecessary technical jargon from their speech, and instead adopt terminology more familiar to the user. They presume that requests for functionality or guidance are well intended, and endeavor to objectively appraise the worth of user's requests in terms of business value rather than personal appeal.

Developers Like To Satisfy A Need, Programmers Like To Finish

Programmers tend to rush headlong into tasks, spending little time considering boundary conditions, low-level details, integration issues and so on. They are keen to get typing as soon as possible, and convince themselves that the details can be sorted out later on. The worst that could happen is that they'll have to abandon what they've done and rewrite it – which would simply be an opportunity to do more coding and perhaps switch technologies as well. They enjoy this trial and error approach, because it keeps activity focused around the coding.

Developers know that the exacting nature of programming means that "more haste" often leads to "less speed." They are also mindful of the temptation to leap into coding a solution before having fully understood the problem. Therefore they will take the time to ensure that they understand the intricacies of the problem, and the business need behind it. Their intent is to solve a business problem, not just to close an issue in a bug tracking system.

Developers Work, Programmers Play

Many software developers enter the work force as programmers, having developed an interest in software from programmer-like, hobbyist activities. Once they learn something of the role that software plays in an organizational context, their sphere of concern broadens to encompass all those other activities that constitute the difference between programmer and developer, as described above.

However, some never make the attitudinal transition from the amateur to the professional, and continue to "play" with computers in the same way they always have, but do so at an employer's expense. Many will never even appreciate that there could be much more to their work, if only they were willing to step up to the challenge and responsibility.

Software engineering, not yet a true profession, places no minimum standards and requirements upon practitioners. Until that changes, hobbyist programmers will remain free to masquerade as software development professionals.

It is the developers that you want working in your organization. Programmers are a dime a dozen, but developers can bring real value to a business. Wise employers know how to tell the difference.

---

[*] First published 9 Oct 2006 at http://www.hacknot.info/hacknot/action/showEntry?eid=90

Management

To The Management*

I am frequently frustrated and disappointed in the standard of management I am subject to. Discussions with my peers in the software industry lead me to believe that I am not alone in my malaise. So on behalf of the silent multitude of software professionals who are disappointed with their management, I would like to remind you - the project manager, team leader or technical manager - of those basic rights to which your staff are entitled.

The Right To Your Courtesy And Respect

Of the complaints I hear directed towards management, the most frequent concern dishonest, abusive or otherwise inappropriate behavior. Remember that no matter how angry or frustrated you may be feeling, it is never okay to direct that anger towards your associates. Intemperate outbursts only engender disrespect and generate ill feeling. As a leader, you are obliged to behave in an exemplary manner at all times.

Respecting your staff implies valuing their opinions, and being prepared to accept their determinations in areas where their expertise is greater than your own. It means acknowledging and accommodating the technical obstacles they encounter, rather than trying to usurp reality with authority.

The Right To Adequate Resources

Skimping on software and hardware resources is an obviously false economy, as a deficit of either impedes your most expensive resource – your people. More commonly overlooked are such environmental resources as lighting, storage space, desk space, ergonomic aids and whiteboards. Workers quickly become dissatisfied if the basic elements of a productive environment are absent.

The resource generally in shortest supply in a software development environment is time. It's not surprising then that unrealistic scheduling is one of the greatest sources of conflict between technical staff and their management. Please keep this in mind - successful scheduling is a process of negotiation, not dictation. Nobody knows more about how long a particular task will take to complete than the person who is to complete it. Your team has the right to be consulted on the scheduling of those tasks they are responsible for, and to be able to meet their commitments without undue stress or hardship.

The Right To Emotional Safety

In many corporate cultures there is a stigma associated with being the bearer of bad news. To ensure that individuals feel safe in expressing unpopular truths, you must not only accept, but also welcome bad news as an opportunity to avert a more serious problem later. Ultimately, you are reliant upon others to keep you apprised of the project's technical progress, so it is obviously beneficial to obtain their insights in uncensored form. For their part, technical staff need to feel that they can openly seek help with their problems, without risk of punitive repercussions.

In a human-based endeavor like software development, mistakes and failures are inevitable. Staff rightfully expects a compassionate attitude from you when dealing with their own failures. When they underestimate a task's completion time, or inject a defect into the code base, they need help in correcting the underlying problem, not castigation for the symptom.

The Right To Your Support

Your staff has the right to expect your assistance in dealing with the issues they encounter. Responsiveness is paramount - issues need to be dealt with in a timely manner, before they can fester into full-blown crises. You must be willing to put aside self-regard and do whatever is necessary to resolve the issue as quickly as possible. This may mean making an unpopular decision, or entering into conflict with other managers. Without the courage and integrity to support your team in this manner, you compromise the well being of the project and the people on it.

Failure to proactively support your team's efforts will necessarily disadvantage them. They have a right to presume you will use your experience and your high level view of the project to forecast the risks they may encounter, and prepare mitigation strategies accordingly.

The Right To Know

Your team expects decisions affecting project staffing, scheduling and scope to be communicated to them quickly and honestly. Unnecessary delays can limit their ability to respond effectively to changing conditions, with consequent stress and time pressure.

Some managers feel they have to shield their subordinates from the political machinations of their organization. This attitude betrays little respect for their team member's maturity, and a basic ignorance of the technical personality, which values painful truths over comforting lies. Your staff has the right to know about anything that impacts on their work, so that they can maximize the chances of achieving their goals.

The Right To Self-Determination

There is nothing so disempowering as to be set goals, but have no control over the means by which one is to achieve them. This is the predicament in which you place your staff if you deny them the flexibility to tailor their work practices to the problem at hand, insisting instead on rigid adherence to methodological or corporate dogma. You may find political safety in playing it by the book, but your people want to work in a way that makes best use of their time and energy, and expect your support in achieving that goal.

It is my recurring observation that management practices that infringe upon the abovementioned rights are common. Equally common is the software professional's lamentation that their management "doesn't have a clue." The two may well be causally related.

So I urge you to put aside your spreadsheets and Gantt charts for a moment and consider the rights of your subordinates. Focus on the basic principles of a humane management style - integrity, respect, courtesy and compassion. Their application cannot guarantee your success as a manager, but their absence will guarantee your failure.

---

[*] First published 12 Jul 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=7

Great Mistakes in Technical Leadership*

"If you are a good leader who talks little, they will say when your work is done and your aim fulfilled, 'We did it ourselves.'" – Lao-Tse, cited in [1]

Perhaps the most difficult job to do on any software development project is that of Technical Lead. The Technical Lead has overall responsibility for all technical aspects of the project – design, code, technology selection, work assignment, scheduling and architecture are all within his purview. Positioned right at the border of the technical and managerial, they are the proverbial "meat in the sandwich." This means that they have to be able to speak two languages – the high-level language of the project manager to whom they report, and the low-level technical language of their team. In effect, they're the translator between the two dialects.

Observation suggests that there are not that many senior techies who have the skills and personal characteristics necessary to perform the Technical Lead role well. Of those I have seen attempt it, perhaps ten percent did a good job of it, twenty percent just got by, and the remaining seventy percent screwed it up. Therefore most of what I have learnt about being a good Technical Lead has been learnt by counter-example. Each time I see a Technical Lead doing something stupid, I make a mental note to avoid that same behavior or action when I am next in the Technical Lead role.

What follows is the abridged version of the list of mistakes I have assembled in this manner over the last thirteen years of watching Technical Leads get it wrong. It is my contention that if you can just avoid making these mistakes, you are well on your way to doing a good job as a Technical Lead. You might consider it a long-form equivalent of the Hippocratic Oath "First do no harm," although given the self-evident nature of many of these exhortations, it is more like "First do nothing stupid."

Mistake #0: Assuming The Team Serves You

Perhaps the most damaging mistake a Technical Lead can make is to assume that their seniority somehow gives them an elevated status in their organization. Once their ego gets involved, the door is open to a host of concomitant miseries such as emotional decision making, defensiveness and intra-team conflict.

I can't emphasize enough how important it is to realize that although the Technical Lead role brings with it many additional responsibilities, it does not put you "above" the other team members in any meaningful sense. Rather, you are on an exactly equal footing with them. It's just that your duties are slightly different from theirs.

If anything, it is *you* that is in service of *them*, given that it is part of your role to facilitate their work. To put it another way, you are there to make them look good, not the other way around.

Mistake #1: Isolating Yourself From The Team

In some organizations, having the title of Technical Lead gives you entitlements that the rank and file of your team do not enjoy. Sometimes, the title is considered sufficiently senior to entitle you to an office of your own, or at least a larger workspace if you must still dwell in cubicle land.

It is a mistake to take or accept such perquisites, as they serve to distance you (both physically and organizationally) from the people that you work most closely with. As military leaders know, it creates an artificial and ultimately unhealthy class distinction between soldiers and officers if the latter are afforded special privileges. To truly understand your team's problems and be considered just "one of the guys" (which you are), you need to be in the same circumstances as they are.

Mistake #2: Employing Hokey Motivation Techniques

Different sorts of people are motivated by different sorts of rewards. Programmers and managers certainly have very different natures, yet it is surprising the number of managers and aspiring managers who ignore those differences and try to reward technical staff in the same way they would like to be rewarded themselves.

For example, managers value perception and status, so being presented with an award in front of everyone, or receiving a plaque to display on their wall where everyone can see it, may well be motivating to them. However programmers tend to be focused on the practical and functional, and value things that they can use to some advantage. Programmers regard the sorts of rewards that managers typically receive as superficial and trite. They have a similar view of "team building" activities, motivational speeches and posters and the like.

So if you want to motivate a developer, don't start cheering "Yay team" or force him to wear the team t-shirt you just had printed. Instead, give him something of use. A second monitor for his computer will be well received, as will some extra RAM, a faster CPU, cooler peripherals, or a more comfortable office chair. It's also hard to go wrong with cash or time off.

Developers are also constantly mindful of keeping their skill sets up to date, and so will value any contribution you can make to their technical education. Give them some time during work hours to pursue their own projects or explore new technologies, a substantial voucher from your local technical book store, or leave to attend a training course that interests them – it doesn't have to be something that bears direct relationship to company work, just as long as it has career value to them.

Mistake #3: Not Providing Technical Direction And Context

A common mode of failure amongst Technical Leads is to focus on their love of the "technical" and forget about their obligation to "lead." Leading means thinking ahead enough that you can make informed and well-considered decisions before the need for that decision becomes an impediment to team progress.

The most obvious form of such leadership is the specification of the software's overall architecture. Before implementation begins, you should have already considered the architectural alternatives available, and have chosen one of them for objective and rationally defensible reasons. You should also have communicated this architecture to the team, so that they can always place the units of work they do in a broader architectural context. This gives their work a direction and promotes confidence that the team’s collective efforts will bind together into a successful whole.

A Technical Lead lacking in self-confidence can be a major frustration to their team. They may find themselves waiting on the Lead to make decisions that significantly effect their work, but find that there is some reticence or unwillingness to make a firm decision. Particularly when new in the role, some Technical Leads find it difficult to make decisions in a timely manner, for they are paralyzed by the fear of making that decision incorrectly. Troubled that a bad decision will make them look foolish, they vacillate endlessly between the alternatives, while their team-mates are standing by wondering when they are going to be able to move forward. In such cases, one does well to remember that a good enough decision now is often better than a perfect decision later. Sometimes there is no choice amongst technical alternatives that jumps out at you as being clearly better than any other – there are merely different possibilities, each with pros and cons. Don't belabor such decisions indefinitely. In particular, don't hand over such decisions to the team and hope to arrive at some consensus. Such consensus is often impossible to obtain. What is most important is that you make a timely decision that you feel moderately confident in, and then commit to it. If all else fails, look to those industry figures whose opinions you trust, and follow the advice they have to give.

Finally, always be prepared to admit that a decision you've made was incorrect, if information to that effect should come to light. Some of the nastiest technical disasters I've witnessed have originated with a senior techie with an ego investment in a particular decision, who lacks the integrity necessary to admit error, even when their mistake is obvious to all.

Mistake #4: Fulfilling Your Own Needs Via The Team

You will occasionally hear people opine that one should not let the personal interfere with the professional. In other words, difficulties at home should not interfere with the execution of duties in the workplace. In some environments, the obvious expression of emotion is simply taboo. But such ideas don't mesh with reality too well. People are holistic creatures and our life experience is not so conveniently compartmentalized, no matter how desirable some Taylorist ideal may be.

Just the same, there are practical and social limitations upon workplace behavior which some may be tempted to flaunt, to the discomfort and embarrassment of their colleagues. The broader one's influence, the greater the opportunity to co-opt activities that should be focused on work, and turn them to personal effect.

For example, meetings (complete with buffet) make a fine social occasion for those not concerned with making best use of company time. Team-building exercises provide an easily excused opportunity to get away from the office and out into the sun, as do off-site training courses and conferences.

Pair programming seems to be most appealing to those who like to chat about their work ... continually. An excessive focus on group consensus-based decision-making for all technical aspects of the project, even the trivial ones, may be a sign that a Technical Lead is more concerned with the sociology of the project and their place amongst it, than with leadership and making efficient use of people's time and effort.

Mistake #5: Focusing On Your Individual Contribution

Changing roles from developer to Technical Lead requires a certain adjustment in mindset. As a developer you tend to be focused upon individual achievement. You spend your time laboring on units of work, mainly by yourself, and can later point to these discrete pieces of the application and say, with some satisfaction, "I did that."

But as a Technical Lead your focus shifts from individual achievement to group achievement. Your work is now to facilitate the work of others. This means that when others come to you for help, you should be in the habit of dropping everything and servicing their requests immediately. A fatal mistake some Technical Leads make is to try and retain their former role as an individual contributor, which tends to result in the Technical Lead duties suffering, as they become engrossed in their own problems and push the concerns of others aside.

The constant alternation between helping individuals with low-level technical problems and thinking about high-level project-wide issues is very cognitively demanding. I've come to call the problem "zoom fatigue" - the mental fatigue which results from rapidly changing between the precise and the abstract on a regular basis. It's like the physical fatigue that the eye experiences when constantly switching focus from long distance to short distance. The muscular effort required within the eye to change focal length eventually leads to fatigue, making the eye less responsive to subsequent demands. Similarly, you get cognitive fatigue when in one moment you are helping someone with an intricate coding issue, and in the next you're examining the interaction between subsystems at the architectural level. The latter requires a more abstract mental state than the former, and alternating between the two is quite taxing.

As a result, people may come to you seeking help with something that has been the sole focus of their attention for several hours or days, and you will find it difficult to "task switch" from what you were just doing into a mindset where you can discuss the problem with them on equal terms. I find it helpful to just ask the person to give me ten minutes to get my head into the problem space, during which I might retreat to my own machine and study the problem body of code in detail, before attempting to help them with it.

Mistake #6: Trying To Be Technically Omniscient

Just because you have the last word in technical decisions, don't think that it is somehow assumed that you are the programming equivalent of Yoda. With the variety and complexity of development technologies always growing, it is increasingly difficult to maintain a mastery of any given subset of that domain. As in most growing fields, those who call themselves "expert" will progressively know more and more about less and less.

It is therefore entirely possible that you will be learning new technologies at the same time as you are first applying them. The mistakes you make and the gaps in your knowledge will be abundantly obvious to your team members, so it is best to abandon at the outset any pretext of having it all figured out.

Be open and honest about what you do and don't know. Don't try and overstate or otherwise misrepresent the extent and nature of your familiarity with a technology, for once you are found out, the trust lost will be very difficult to regain.

There is an opportunity here to widen the knowledge and experience of all team members. You might like to appoint certain people as specialists in particular technologies, giving them the time and task assignments necessary to develop a superior knowledge of their assigned area. To avoid boredom and unnecessary risk, be sure to give these resident experts plenty of opportunity to spread their knowledge around the team, and to exchange specialties with others.

Adopting this "collection of specialists" approach makes it clear that you are not presuming to be all things to all people; and that you have faith in the abilities of your colleagues. But it will require you to park your ego at the door and be prepared to say "I don't know" quite frequently.

But be careful not to lean on others *too* heavily. It is still vitally important for you to have a good overarching knowledge of the technologies you are employing, particularly those elements of them that are critical to their successful interoperation in service of your system’s architecture.

Mistake #7: Failing To Delegate Effectively

To successfully lead a group, there must be an attitude of implicit trust and assumed good intent between the leader and those being led. Therefore a Technical Lead must be willing to trust his team to be diligent in the pursuit of their goals, without feeling the need to watch over their shoulder and constantly monitor their progress. This sort of micromanagement is particularly loathed by programmers, who recognize it as a tacit questioning of their abilities and commitment.

But ineffective delegation can also arise for selfish reasons. Several times now I've seen Technical Leads who like to save all the "fun" work for themselves, leaving others the tedious grunt work. For example, the Technical Lead will assign themselves the task of evaluating new technologies, constructing exploratory and "proof of concept" prototypes, but once play time is over and the need for disciplined work arrives, hand over the detailed tasks to others.

Not only is effective delegation desirable with respect to team morale and project risk, on large projects it is simply a necessity, as there will be too much information to be managed and maintained at once for one person to be able to cope.

Mistake #8: Being Ignorant Of Your Own Shortcomings

Some people simply don't have the natural proclivities necessary to be good Technical Leads. It's not enough to have good technical knowledge. You must be able to communicate that knowledge to others, as well as translate it into a simpler form that your management can understand.

You also need good organizational skills. Coordinating the efforts of multiple people to produce a functionally consistent outcome is not easy, and demands a methodical and detail-oriented approach to planning and scheduling. If you can't plan ahead successfully, you will find yourself constantly in reactive mode, which is both stressful and inefficient.

If you don't have these qualities naturally, you may be able to develop them to some extent, through training and deliberate effort. But it may ultimately be necessary for you to lean on others in your team to support you, should they have strengths in areas in which you have weaknesses.

Mistake #9: Failing To Represent The Best Interests Of Your Team

Perhaps the most nauseating mistake a Technical Lead can make is to become a puppet of the management above them. As the interface between management and technicians, it is the Technical Lead's role to go into bat with their management to represent the best interests of their team. This means standing up to the imposition of unreasonable deadlines, fighting for decent tools and resources, and preventing the prevarications of management from disturbing the rhythm of the project. A weak-willed or easily manipulated Technical Lead will incur the disrespect of his team.

Unfortunately, such spineless behavior is quite common amongst the ranks of the ambitious, and you don't have to look far to find obsequious Technical Leads who will gladly promise the impossible and impose hardship on their team, in the interests of creating a "can do" image for themselves.

Mistake #10: Failing To Anticipate

An essential part of the Technical Lead's role is keeping an eye on the "big picture" – those system-wide concerns that are easily forgotten by programmers whose attention is consumed by the coding problem they currently face.

These "big picture" issues include those non-functional requirements sometimes called "-ilities" - maintainability, reliability, usability, testability and so on. If you don't make a conscious effort to track your progress against these requirements, there is a high probability of them slipping through the cracks and being forgotten about until they later emerge as crises.

If you don't have a dedicated project manager, it may also fall to you to handle the scheduling, tracking and assignment of tasks. It isn't uncommon for Technical Leads to find themselves playing dual roles in this manner. You may not be very fond of such "administrative" duties, but their efficient performance is critical to the smooth running of the project, and for the developers to know where they are and where they're going. Don't make the mistake of ignoring or devaluing these tasks simply because they are non-technical in nature.

Mistake #11: Repeat Mistakes Others Have Already Made

It is common for developers to dismiss the experience reports of others as having no relevance to their own situation. Indeed, it is wise to approach all anecdotal evidence with skepticism. But it is unwise to *completely* disregard the advice of others, particularly when it is accompanied by sound reasoning, or can be independently verified. Ignoring good advice can be very expensive; as Benjamin Franklin said, "Experience keeps a dear school but fools will learn in no other."

The unwillingness of developers to learn from the mistakes of others, and the ease with which you can encounter software project horror stories in the literature and recognize your own projects in them, is evidence suggesting that the software industry as a whole is not getting any wiser.[2] You need not contribute to that collective stupidity.

Mistake #12: Using The Project To Pursue Your Own Technical Interests

Remarkably, developers can reach quite senior levels in their organization without having learnt to appreciate the difference between work and play. Many are attracted to programming to begin with because, as hobbyists, they enjoyed fooling around with the latest and greatest technologies. Somehow they carry this tendency to "play" with technologies into their working lives, and it becomes the aspect of their jobs that they value most. From their perspective, the purpose of a development effort is not to create something of value to the business, but to create an opportunity to experiment with new technologies and pad their CV with some new acronyms.

Their technology selection is based upon whatever looks "cool". But a rational approach to technology selection may yield quite a different result to one guided by technical enthusiasm or a fascination with novelty. New technologies are often riskier choices, as the development community has not had much time to apply the technology in varying circumstances and thereby discover its weaknesses and shortcomings. Putting an immature technology on a project's critical path is especially risky. So an older, tried and true technology may be a more rational choice than a new, unproven one.

Mistake #13: Not Maintaining Technical Involvement

In order to fully appreciate the current status of the project as well as the difficulties your team is facing, it is vital that you maintain a coding-level involvement in the project. If you're not cutting code, it is too easy to become divorced from the effects of your own decision making, and to be seen by other developers as being out of touch with the technical realities of the project.

Mistake #14: Playing The Game Rather Than Focusing On The Target

In some organizations, being a Technical Lead is a politically sensitive position. Technology choices, work assignments and project outcomes are all just tools to be used in the pursuit of personal agendas. To some, this "game" of political influence is both fascinating and addictive. They play it in the hope of gaining some advantage for themselves, and do so to the detriment of the project and the individuals upon it. When they don't have their eye on the ball like this, devoting more energy to Machiavellian maneuverings than to the technical difficulties of the project, then the project inevitably suffers.

Mistake #15: Avoiding Conflict

Many people find interpersonal conflict distasteful. Some dislike it so much that they will do practically anything to avoid it, including giving up in technical disputes. Such people are prone to being walked over by those more aggressive and forthright.

This is bad enough for the individual, but worse if that person is meant to be representing the best interests of a team. A meek Technical Lead can be a real liability to a development team, who will find themselves buffeted about by external forces that they should have been shielded from, and burdened by demands and goals that are not informed by the project's reality.

With such a disposition, a Technical Lead may be unable to even deal effectively with unruly behavior or inadequate performance from members of their own team.

Mistake #16: Putting The Project Before The People

It's one thing to be focused on the project's goals, but quite another to adopt a "succeed at all costs" attitude. Ambitious Technical Leads, concerned with the image they project to their management, sometimes accept impossible goals or unreasonable demands, because they lack the courage or integrity to say "no." These goals then become the development team's burden to shoulder, leading to increased stress, higher defect injection rates, longer working hours and lower morale. There is a tendency to be so focused on the end goal that the effects of the project on the developers gets overlooked. It is not uncommon for successful delivery on a high pressure project to be followed by the resignations of several disgruntled team members, making the project's triumph a pyrrhic victory indeed.

Given the costs of hiring and training staff, treating developers as expendable resources makes no financial sense, quite aside from the ethical implications of such treatment. A wise Technical Lead will know that putting the well-being of the developers first also produces the best results for the project and the business. Project success should leave the participants satisfied with their achievement, not burnt out and demoralized.

Mistake #17: Expecting Everyone To Think And Act Like You

Being a Technical Lead may be the first time you are exposed so frequently and directly to the problem solving styles and low-level work habits of others. Programming is traditionally an individual activity. Programmers are often able to face the technical difficulties of their work in isolation, emerging sometime later with the completed solution. But as a Technical Lead you will frequently be called on to help those who are stuck part way through the problem-solving process, unable to proceed. Seeing a solution that is "under construction" might be a bit of a shock to you at first, as you may find your colleagues approach to problem solving dramatically different to your own. Some people work "outside in", others "inside out", others jump all over the place, some work quickly with lots of trial and error, others slowly and methodically. It is tempting to stand in judgment of approaches and methods that don't gel for you, pronouncing them somehow inferior. Avoid the temptation. Learn to accept the varieties of cognitive styles on your team, and recognize that this cognitive diversity may actually be an asset, for the variety of perspective it brings.

Mistake #18: Failing To Demonstrate Compassion

Although I've put this last, it is in some ways the most important of all the mistakes listed here. Always remember that your team members are people first and programmers second. You can expect them to be temperamental, inconsistent, proud, undisciplined and cynical – perhaps all in the same day. Which is to say they are flawed and imperfect, just like you and everyone else. So cut them some slack. Everyone has good and bad days, strengths and weaknesses; so tolerance is the order of the day.

If someone breaks the build, it's no big deal. If a regression is introduced, learn something by finding out how it got there, but don't get upset over it or attempt to assign blame. If a deadline is missed, stand back from the immediate situation and appreciate that in the grand scheme of things, it really doesn't matter. Mistakes happen and you should expect your colleagues to make many, as you will surely make many yourself.

[*] First published 11 Jun 2006 at http://www.hacknot.info/hacknot/action/showEntry?eid=87
[1] *Becoming A Technical Leader*, G. M. Weinberg, Dorset Hourse, 1986
[2] *Facts and Fallacies of Software Engineering*, Robert L. Glass, Addison-Wesley, 2003

The Architecture Group*

An organizational antipattern that I have seen a few times now is the formation of an Architecture Group. Architecture Groups generally have the following purposes:

- To design the enterprise architecture shared by a group of applications within an organization
- To review the design of projects to ensure they are consistent with the enterprise architecture
- To prescribe the standard technologies to be used across projects in the organization

In summary, the Architecture Group is an internal "governing body" and "standards group" rolled into one. Membership of the group tends to be restricted by seniority – the architects and senior technical staff.

In general, the Architecture Groups I've witnessed in action have been disastrous. That's not to say that it need necessarily be so – I have no legitimate basis for generalizing beyond my direct experience – but based on the reasons that I've seen these groups fail, I conject that failure is a likely outcome of any such group.

The negative impact of an Architecture Group often originates from the tendency to create an "us and them" mentality amongst staff. Because the group makes technology and design decisions which are then imposed upon other projects, those working on individual projects come to resent the architecture group for the constraints they have placed upon the project. Working at the overview level, as an architecture group does, it is difficult or impossible to keep track of the low level details of a variety of projects. And yet the details of those projects are key determinants of the suitability of the technologies and designs that the architecture group deals with. Project staff come to view the architecture group as dwelling in an ivory tower, from where they can afford to overlook the troublesome aspects of the projects in their influence.

Members of the architecture group can begin to share this view. They consider their decision making more objective and sensible precisely because it is not influenced by the low level concerns of individual projects. Once high level consideration has occurred, any difficulties encountered while implementing those decisions are dismissed as "implementation details" that are beneath the group's level of concern.

The major source of trouble with architecture groups seems to be the social dynamic that builds up around them. They have a tendency to become a clique that is in overestimation of its own collective abilities, because it is deprived of any negative feedback concerning the consequences of the decisions it makes. The absence of feedback results in part from the unwillingness of project staff to criticize those senior to them, and in part of the self-imposed isolation of the architecture group, which makes its decisions from behind closed doors.

The issue of seniority is a real stumbling block, because senior staff may have great difficulty in admitting that they have made a poor decision, even when it is perfectly obvious to project staff that this is the case. Any adjustment to the decrees of the architecture group, once made, results in a perceived loss of face which the members of the architecture group can ill afford. Being senior, they are perhaps more cognizant of the political forces at work in the organization. Perhaps they are more ambitious, and therefore reticent to concede wrong doing for fear of the impact it might have on their reputation. Perhaps they view the objections of project staff as a challenge to their authority. In any case, members of the architecture group develop an ego identification with the decisions they make, which leads them to ignore or devalue negative feedback from project staff – leading to the reinforcement of the architecture group's external image as being isolated from the project community.

Consider also that people working in architectural roles tend to be abstractionist by nature. They are comfortable working at a high level and just trusting that the low level details will work themselves out. When project staff object that a decision made in the abstract has resulted in concrete difficulties at the implementation level, the abstractionist is prone to characterizing the situation as one of a well conceived plan that has been fumbled in the execution. In other words, they shoot the messenger, preferring to blame the implementation of their decision rather than the decision itself, which is perfect – as long as it is only considered in the abstract.

Conclusion

Those who institute an architecture group in their organization may be courting disaster. There is a strong tendency for the group to become cliquish, divorced from the consequences of its decision making, and the object of wide-spread resentment within the organization. Coordination of projects and adherence to enterprise architectures should occur in a way that does not impinge upon individual project's chances of success, nor rob them of the ability to solve the particular problems of their project in an effective way.

[*] First published 29 Mar 2005 at http://www.hacknot.info/hacknot/action/showEntry?eid=73

The Mismeasure of Man*

Software developers are drawn to metrics for a variety of reasons. Generally, their motivations are good. They want to find out something meaningful about the way their project is progressing or the way they are doing their job. Managers are also drawn to metrication for a variety of reasons, but their motives are not necessarily honorable. Some managers view metrics as an instrument for getting more work out of their team and detecting if they are slacking off.

Performance metrics – metrics intended to quantify individual or group performance – can be useful if they are employed sensibly and in full awareness of their limitations. Unfortunately, it is very common for performance metrics to be gathered and interpreted in ways that are ultimately harmful to a project and its developers. Many is the metrics program that, through inept implementation and application, has engendered anger and resentment amongst those it was intended to benefit.

Below, we consider various performance metrics commonly encountered in development environments, the ways they are abused, and illustrate their misuse with some examples taken from my own experience and the experience of others as they have related it to me.

The Number Of The Counting

Face Time

This is perhaps the most commonly abused "metric" in the software development world. For reasons of both tradition and convenience, many managers and developers alike persist in considering the number of hours spent in front of the screen as being some indication of how devoted a programmer is to their work. Those that work long hours are considered "hard workers," those that keep regular hours are considered "clock watchers."

The fault behind such thinking is the assumption that software development is a manufacturing-like process, rather than a problem-solving process. If a worker on a production line works an extra hour then the result is an extra hours' worth of stuff. If they work an extra three hours then the result is an extra three hours worth of stuff; which will be exactly three times the quantity of extra stuff they would've produced had they only worked a single extra hour. If their role on the production line is menial assembly work, then the quality of the stuff they produce in their third hour of overtime will be the same as the quality of the work from their first hour of overtime. In such an environment, it is reasonable to see productivity as a direct function of time on the job.

But software development is nothing like this mechanistic process. It is a complex, intellectual effort conducted by knowledge workers, not a menial assembly task performed by laborers. So more hours spent in front of the screen does not necessarily equate to more progress. For example, long work hours might be a result of problems such as:

- Relying on trial and error rather than thinking ahead
- Goofing off surfing the web or socializing
- Solving the wrong problem, and having to start again
- Gold-plating (extending scope beyond what is required, simply for the satisfaction of it)
- Using a lengthy, inefficient algorithm rather than a smaller, elegant one
- Writing functionality that should have been purchased in a third party library
- Making the solution more generic than is necessary
- Poor understanding of the technologies employed, resulting in a lot of thrashing
- Losing a lot of time to debugging, because of the higher defect injection rates that occur when working while fatigued
- Overly ambitious scheduling resulting from poor self-insight and lack of experience

So by expecting or encouraging long working hours, we may simply be rewarding poor performance and inefficient work practices.

I first encountered the obsession with working hours at a small "dot com" company I once had the misfortune to work for. Full of bright and enthusiastic young people, the CTO of this company considered his stable of go-getters a resource to be exploited to the fullest. Not being the most technically aware of CTOs he was unable to assess the performance of the technical staff that reported to him in any meaningful way, so he was forced to rely on what he considered to be secondary indicators of performance – the number of weekly hours each employee logged in their electronic time-sheet.

Those with more experience of his somewhat indirect approach to assessment were quite generous when it came to such time-keeping tasks, logging some spectacular hours – some of which they actually worked. Those unfamiliar with the man's chronological obsession, such as myself, made the mistake of working efficiently and recording their work hours accurately. This did not go down so well.

In my letter of resignation I cited unscrupulous and irrational management practice as one of the principal reasons I was leaving. On my last day at said company I received what is, to date, the only written response to a resignation that I have ever encountered. The response contained a month-by-month tabulation of average daily working hours – both the company average and my personal figures. Of course, my "performance metric" was disgustingly normal, whereas the company averages seemed to indicate that many staff were dedicating all their waking hours to work. The conclusion was obvious – I was not putting in the sort of effort that was expected of me. How right they were.

Lines Of Code

It should be common knowledge that lines of code (LOC) and non-comment lines of code (NLOC) are not measures of size, productivity, complexity or anything else particularly meaningful. It is none-the-less very common to find them being used in the field to quantify exactly these characteristics. This is probably because these metrics are so easily gathered and there is an intuitive appeal to equating the amount of code written with the amount of progress being made.

But it is a big mistake to consider large quantities of code necessarily a good thing, for large volumes of code may also be symptomatic of problematic development practices such as:

- Unnecessarily complex or generic design
- Cut-and-paste reuse
- Duplication of functionality

Large quantities of code can also bring such problems as:

- A greater opportunity for bugs
- A greater maintenance burden

- A greater testing effort
- Poor performance

So by rewarding those who produce larger quantities of code, we may simply be encouraging the production of a burdensome code base.

The story is told of a team of developers whose well-meaning but uninformed manager decided that he would start measuring their individual contributions to the code base by counting the number of lines of code each of them wrote per week. Fancying himself as more technically informed than most other middle managers, he wrote a simple script to count the number of lines of code in a file.

The project was written in C. Figuring that most statements in C ended in a semicolon, he presumed that his script could just count the number of semicolons in the file and that would give him the number of C statements. He congratulated himself on thinking of this clever counting method, which would not be susceptible to differences in coding style between developers, nor any of the techniques developers sometimes employed to try and manipulate metrics in their favor by changing the layout of their code.

However a few of the developers got wind of the technique their manager was using, and started writing function comments containing long rows of semicolons to delineate the beginning and end of the comment block.

Their measured rate of code production skyrocketed ... so much so that their manager became suspicious and, looking at the code to manually verify that his script was working correctly, discovered what was going on. But the developers simply claimed that their recent change in comment style was just an innocent search for greater code readability The manager could not prove otherwise.

Function Points

In some circles, Function Points (FPs) have currency as a way of measuring the size of a piece of software. There are complex counting procedures that enable functionality to be expressed as a number of FPs in an ostensibly language-independent way. The formation of the IFPUG (International Function Point Users Group) and the amount of semi-academic study they have received has invested FPs with a certain amount of faux credibility. However, this credibility is undeserved, as FPs are a fundamentally flawed metric. They are not a valid unit of measurement, nor can they validly be manipulated mathematically. Any metric involving them is approximately meaningless. FPs have been discussed at length in a previous article[1].

Screens

Having worked principally in the area of rich-client and desktop applications, I've witnessed numerous mismeasures of progress from this domain. The most foolish of them was to use a "screen" (dialog / window) as a unit of measurement. Thus, if programmer A implemented two dialogs in the time programmer B implemented one, A was considered to be twice as productive as B.

The faults with such an approach are alarmingly obvious, but often ignored by an unthinking management that is too impressed by the fact that they can attach numbers to something, which creates a false impression that they are measuring something. Such are the perils of metrication in the hands of the ignorant.

To labor the obvious, here are a few reasons one programmer might produce more "screens" than another, that have nothing to do with productivity:

- Their screens were simpler in appearance and/or behavior.
- Their screens were sufficiently similar in appearance and/or behavior, so there could be code re-use between them.
- Their screens could be constructed with standard GUI components, without the need for custom components being developed.
- Their screens were not the end result of a usability-based design process, but were whatever was most programmatically expedient.

By counting "screens" as a measure of progress, we encourage programmers to race through their tasks, giving short shrift to issues of usability and reuse.

I once worked for a small firm in the finance industry. Their flagship product was a client/server application for managing investment portfolios. I was brought in, together with another GUI guy, to extend the functionality of the system and clean up a few of the existing screens and dialogs. Under the hood, this product was a disaster. Poorly coded, undocumented and architecturally inconsistent, it was the end result of the half-hearted, piece-meal hacking of many previous generations of contractors.

The gentleman who had shepherded all these contractors through the company doors, and who considered himself both Technical Lead and Project Manager, was not heavily into software. Indeed, he never actually bothered to look at the application's code. He had only one way to gauge individual or collective progress and that was on the basis of appearance. If a piece of work involved lots happening on the screen, then he figured that it represented a lot of work. If it wasn't visually significant, then he figured there probably wasn't much to it. Let's call him Senior Idiot.

He and I did not get on so well, right from the start. I'm told I don't suffer fools lightly and as fools go, this guy was an exceptional specimen. My fellow GUI guy was no better. Examining the code that he wrote and the work he delivered, it was clear he was working at a level consistent with the noxious quality of the existing code base. Let's call him Junior Idiot.

A few months after I started, Big Idiot took me aside and asked why my progress was "so slow." I thought this was an interesting comment, given that by my own analysis I was generating good quality code at a rate several times the industry average. Both the code and the resulting interfaces were some of the best they had in the entire, sorry product. When I enquired how he had determined my progress was "slow" given that he never actually looked at code, he explained that he was comparing the "number of screens" Little Idiot had managed to grunt out, to what I had developed in the same time. Little Idiot was some way in front.

He was correct. Little Idiot had produced several rather large screens (large in the sense that they occupied many pixels, not in the sense that they represented a lot of functionality). They were usability disasters, every one of them, and the product of some pretty deft cut-and-paste but, scatological in quality as they were, they were there to be seen.

After some chuckling, I tried to carefully explain to him the "discrepancy" that he saw was because Little Idiot was spitting out rubbish as quickly as possible, and I was taking some time to do a decent job. Additionally, Little Idiot was producing non-reusable code , whereas I was writing general purpose code, reuse of which would mean that future work, both my own and others, would progress much more quickly than Little Idiot could ever do. He was not convinced and my time at this little company came to an end shortly thereafter, much to our mutual relief.

Iterations

Unbelievable as it is, I can honestly say that I've seen entire projects compared on the basis of what iteration they are up to in their respective schedules. Suppose projects A and B both employ an iterative methodology. A is in the third of five planned iterations, B is in the fourth of seven planned iterations. Some observers may then conclude that project A is behind project B because "three" is less than "four." Others might conclude that project A is ahead of project B because it has completed 60% of its iterations and B only 57%.

I recall the organization in which I first encountered this. A rather hubristic, research oriented environment in which some very clever people worked. Sadly, the quality of the management was not on a par with the quality of the technical staff. As they say, "A fish rots from the head down," so it was no surprise that the manager at the top was not as clued up in many areas as one might like.

At this time, "data warehousing", "knowledge management", "project cross-fertilization" and "knowledge repositories" were the buzzwords that substituted for critical thought. Mashing all these concepts together in his head, the top guy decided to establish a "project wall" in the office, upon which the project managers were required to post the Gantt charts for their respective projects, and keep them up to date. This strategy was meant to promote some sort of comparison and knowledge sharing between projects, although exactly how this was to be done meaningfully was never quite made clear. The device became widely known as "The Wall Of Shame", as that was its obvious but unstated purpose – to publicly shame those managers whose projects were running behind schedule. Presumably, the potential for embarrassment was meant to encourage individual project's to maintain schedule.

It came as a surprise to no-one but the man who instituted the scheme, that it had precisely no effect on anything, except to become the focus of widespread derision.

Tasks / Bugs

Many software development teams allocate work to individuals on a per-task basis. Typically, these tasks are tracked in some electronic form – perhaps as bugs in a bug tracking system or tickets in a trouble ticket system. XP projects like to track tasks on pieces of card because the arts-and-crafts association creates the illusion of simplicity (an illusion which disappears when reports of any kind are required, or when the first strong breeze comes along).

Regardless of the mechanism used, "the task" is so useful as a unit of work allocation that it is very tempting and convenient to think of it as a unit of measurement. Of course, it is not a unit of measurement, as no two tasks are the same. A tiny, one-line bug fix might be captured as one task, as might the implementation of an entire subsystem. The granularity is ever-varying, making any mathematical comparison of task counts meaningless.

But convenience outweighs reason and so one frequently finds, particularly amongst the ranks of management, the tendency to equate high rates of task completion with high productivity and effort, and lower rates with lower productivity and effort. The mistake is so common that developers become quite practiced at gaming the system to make themselves look good. Common image enhancement techniques include:

- Breaking work down into unusually small tasks, thereby enabling a greater number of tasks to be completed at a faster rate.
- Registering tasks as completed before they have been properly tested. This enables bugs to be readily found in the work, each of which will be considered a separate task. These tasks can be completed relatively quickly because the programmer is familiar with the code at fault, having just written it.
- Registering tasks multiple times, describing it in slightly different ways each time. Once completed, all the tasks are closed, with all but one marked as duplicates. If the management forgets to exclude duplicate tasks from their reporting, the programmer's rate of task completion is artificially inflated. He might also "forget" to mark some of the duplicate tasks as being duplicates, to further enhance the effect.
- When a task is found to be more involved than originally thought, rather than revise the scope of the existing task, new tasks are spawned to capture the unanticipated work. Their eventual completion will mean that the number of "completed" tasks registered against the programmer's name is greater.
- When selecting work to do, programmers gravitate towards the short tasks which can be easily dispensed with, enabling them to quickly get runs on the board.

When invalid metrics are gathered, the result is often to contort the team member's work practice so as to create the best perceived performance, regardless of what their actual performance might be.

A colleague once related to me the story of two teams of developers in a multinational company who reported to the same manager. One team contained three developers working mainly on maintenance tasks, documentation and bug fixing. The other, containing six developers, worked on per-client product customizations. Both happened to use a common issue tracking system.

A developer from the smaller team complained to the manager about the discrepancy in work loads between the two teams. He felt that his own team was dreadfully overburdened while the larger one just seemed to be taking it easy. Although uncertain that the developer's complaint was valid, the manager felt compelled to "handle" the situation in a managerial kind of way. Turning to the issue tracking system he did a few simple queries and discovered that the small team was closing issues at nearly twice the rate of the larger team. This struck him as confirmation of the developer's complaint. After all, a team twice as large should be getting through issues much faster than a team half its size.

So the manager sent an e-mail to all members of both teams, and CC'd the general manager. In this e-mail he highlighted the discrepancy in issue closure rate for the two teams, chastised the larger team for slacking off and praised the smaller team for their hard work.

The original complainant was suitably appeased, but the other members of his team, along with the entirety of the larger team, were not quite so happy. The following day, the leader of the larger team came to the managers office and explained to him, in a tone of barely suppressed hostility, that the two teams worked on completely different sized issues, and so comparing issue closure rates across the two was quite meaningless. The smaller team addressed issues that could generally be resolved in a single day, two days at the most, and so naturally they got through them at a fairly rapid pace. His team, the larger one, addressed implementation issues that might legitimately involve weeks of effort, including design, requirements gathering and testing. He was more than a little offended that his hard working team was being reprimanded on such an irrational basis.

The manager admitted his error – but of course, never apologized to those he had offended.

Version Control Operations

Astonishing as it may seem, some developers like to commit changes to their version control system frequently to create the impression that they are hard at work. This only works if you are managed by the technically incompetent. In other words, it works more frequently than you would like.

Requirements Completed

Regardless of whether you capture your requirements in tabular, use case or story card format, individual requirements make spectacularly bad units of measurement.

Consider the enormous variation in scope that can exist between one requirement and another. "The user shall not be able to enter an age greater than 120 or less than 0" counts as "one requirement"; so does "The system shall reserve the section of track for the given vehicle in accordance with safe-working procedure SP-105A." But the latter is probably a far greater undertaking than the former, and we would expect it to take significantly more time and effort to complete. Pity the developer who is assigned the task of satisfying this requirement, only to have his labors viewed as an achievement "equal" to that of his colleague who was assigned the simpler age-related requirement.

Noise Generated

Some programmers just get the job done. Others seem to find it necessary to let others know that they are getting the job done. You've probably met the type before. Every little obstacle and difficulty they encounter seems to be a major drama to them – almost a theatric opportunity. These are the same programmers who will work overtime to fix problems of their own creation, then seek credit for the extra hours they've put in. Although there is no number associated with their vociferations, they effectively multiply the amount of perceived work they are doing, and inflate the perceived effort they are making by drawing attention to their actions.

I once worked with such a programmer. He was a hacker of the first order; and I use the word "hacker" in the pejorative sense. Each day over the lunch room table he would regale us with stories of his mighty development efforts, the technical heights to which he had scaled, and the complex obstacles he had overcome – all of these adventures apparently having happened since the previous day's story-telling episode. But when you actually looked in the source code for evidence of these mighty exploits, you would find only an amateurish and confused mess, and be left wondering how so much difficulty could have been encountered in the achievement of such modest results.

Pages Of Documentation

Used intelligently, documentation makes a useful component of the development process. But when seen as an end in itself, documentation becomes a time-consuming ritual for comforting self-serving administration. Strange then that we should so frequently see, most often in heavily bureaucratic environments, people striving to generate technical specifications that are as voluminous as possible, apparently fearing that brevity will be interpreted as evidence of laziness. A page fails to measure either effort or progress for all the same reasons that "Lines of Code" fails. Stylistic variations mean there is little relationship between volume of text and effective communication as there is between volume of code and functionality.

Conclusion

In the above you will have noticed the same problems occurring again and again. All these scenarios reflect a poor understanding of the basics of measurement theory, together with a willingness to rationalize a metric's invalidity because of the ease with which it can be collected.

Essentially, a valid unit of measurement is a way of consistently dividing some real world quantity into a linear scale. In other words, X is a valid unit of measurement if X is half as much of something real as 2X is, one third as much of something real as 3X, and so on. For this to be true, all instances of X must be the same. For example, the "meter" is a valid unit of measurement because 2 meters is twice the linear distance of 1 meter, and all instances of the "meter" are the same. The "1 meter" that exists between the 0 and "1 meter" marks on your tape measure is the same quantity of something real as the "1 meter" between the "4 meters" and "5 meters" marks. Compare this to an invalid metric like a "task." A task doesn't divide any real world quantity into equal portions. In particular, it doesn't divide effort or work into equal portions, because different tasks might require different amounts of work to complete. So "2 tasks" is not twice "1 task" in any meaningful sense. Put more simply, when comparing tasks, you're not comparing like with like.

The attraction to metrics, even false ones, perhaps stems from the false sense of control they offer. Once we pin a number on something, we feel that we know something about it, that we can manipulate it mathematically, and that we can make comparisons with it. But these statements are only true for valid metrics. For false metrics like bugs, tasks, function points, pages, lines of code, iterations etc., we create only the illusion of knowledge. The illusion may be comforting, particularly to those of an analytical bent, but it is also an invitation to misinterpretation and false conclusions.

We might try and rationalize these invalid metrics, figuring that they may not be perfect, but they are "close enough" to still have some significance. But really this is just wishful thinking. You might think, "our tasks may not be exactly the same, but they're close enough in scope that 'tasks completed' still means something." Really? What evidence do you have that these tasks are of approximately equal scope? If you're honest with yourself, you'll find you've got nothing more than gut feel to justify that statement. Yet the very reason we use metrics is to obtain greater surety than that provided by gut feel. So we see we are really just trying to convince ourselves that our own guesswork can be somehow made better by hiding it behind a number – borrowing the credibility often associated with quantification.

Metrics are a tool easily abused. A common cause of mismeasurement is their punitive application with the intent of motivating higher productivity. In their zeal to find some way to meet a deadline, managers sometimes sacrifice reason for expediency, hoping that some hastily contrived metric can be used to convince someone that they need to be working harder. Of course, such tactics frequently backfire, resulting only in developers feeling resentful of such numeric bullying.

---

[*] First published 6 Aug 2006 at http://www.hacknot.info/hacknot/action/showEntry?eid=88
[1] See Function Points:
Numerology for Software Developers

Meeting Driven Development*

The software development arena is the land of the perpetual "me too." Populated by an eager community of "joiners," every band wagon that comes along is soon laden down by a collection of hype merchants who, recognizing the next big thing when they see it, are keen to milk it for all it is worth. Extreme Programming – that marketing campaign in search of a product – was a particularly fruitful source of commercial spin-offs. When Extreme Testing, Extreme Database Design, Extreme Debugging and Extreme Project Management had run their course; when XP's agile prequel had fostered a small industry based on old saws spruced up with a few neologisms; those looking to make a name for themselves turned to another member of the XP franchise – Test Driven Development – for entrepreneurial inspiration.

TDD: The Progenitor Of MDD

If you have not read Kent Beck's insufferable tome "Test Driven Development,”[1] let me spare you the time and insult by presenting the expurgated version here:

Hello boys and girls. Once upon a time there was a thing called Test Driven Development – it looked for all the world like an impoverished rendering of Design by Contract[2] *only much cooler.*

The ditto brigade latched onto TDD and got to work. We soon had, sprouting like weeds from between the pavement stones, "*Blah* Driven Development", for all conceivable values of *Blah*. It became *de rigueur* to have something driven by something else. Not since Djikstra's "Goto Statement Considered Harmful" had there been such a rash of imitation.

The appeal of such development models is in the simplistic and unrealistic view that a complex activity can be reduced to consideration of, or focus upon, a single factor. But software development is an inherently multivariate process requiring intelligent compromise between competing forces. Unfortunately, such a view is hard to sell.

The fantasy is more appealing ... focus on *blah*, make it the basis of your development effort, and the rest will fall into place as a natural consequence. If you can convince yourself that *blah* is analogous to a set of requirements or an abstract model then you can also dispense with the unpleasantness of requirements elicitation and design. With sufficiently zealous adherence to *Blah*DD, combined with a healthy dose of metaphor and supposition, the formerly complex and uncertain undertaking of developing a piece of software turns into the routine application of a silver bullet. Or so some would have you believe.

Such "one stop" philosophies are a recipe for disappointment, but will no doubt continue to sell well, for the same reasons that "get rich quick" and "lose weight fast" schemes do – the promise of an easy fix.

To show how it's done and perhaps make an obtuse point or two, let's look at the latest *blah* to exhibit in the software development road show – Meeting Driven Development.

An Introduction To MDD

MDD is more than an approach to software development, it is a cultural force. If you're lucky, you are already working in an environment conducive to the meeting mindset. In some corporate cultures meetings are so endemic that they have become an integral part of the corporate identity. For example, an IBM insider tells me that most staff consider IBM to stand for "I've Been to Meetings".

If your corporate culture is not so amenable to MDD, do not despair. You can surreptitiously introduce it into your project without much effort and when others see how successful you have been, it will quickly spread through the rest of your organization like a virus.

I suggest you begin by creating a localized "meeting zone" in your project area. Put a table and some chairs right in the middle of your project's work area, so that project staff need only turn their chairs around and wheel them a short distance in order to assume the meeting position. You will enjoy the disgruntled mutterings of nearby programmers as they struggle to concentrate amidst the noise such meetings create.

The only practical skill MDD entails is the ability to recognize and achieve *meeting mode*. Meeting mode is the colloquial name for what is more properly known as *corporate catatonia* – the mental state achieved by those meeting attendees who cannot or will not participate, instead turning their attention inward. MDD veterans describe the state as being peaceful, meditative and excruciatingly dull. Some claim to have undergone "Out of Body Corporate" experiences while in deep states of meeting mode, during which they separate from their physical bodies, leave the meeting room and go on annual leave.

External indications that an MDD practitioner is in meeting mode include:

- Vacant staring into the middle distance.
- Methodical doodling upon note paper.
- Slowing or cessation of respiration.
- Extended periods of silence.

Types Of Meetings

In MDD, we encourage the use of meetings at every opportunity and for every purpose. Our motto is "Every Meeting Is a Good Meeting". While you can hold a meeting for almost any purpose that comes to mind, there are certain types of meetings that tend to feature commonly in software development environments. It is important that you develop some facility with each of them.

Type #1: The Morning Stand-Up Meeting

You should begin the day with a team meeting, and in this respect MDD is in agreement with XP's practice of holding daily "stand-up" meetings. Like many meetings that are driven by the calendar rather than by a need, your morning meeting will probably devolve into a pointless ritual that serves only to give the organizer a sense of control and influence. For those desperately trying to fulfill a management or leadership role, but lacking the basic proclivities that such roles demand, these ritualistic meetings can also help sustain their delusions of competence, as holding and attending meetings seems like a very managerial thing to do.

Type #2: The Requirements Meeting

A typical requirements meeting involves some technical staff and stakeholders sitting down to discuss the functional requirements for a unit of work. If there are any questions concerning requirements previously elicited, they are tabled here. It is a chance for potential users to lobby technical staff and their managers for the inclusion of their favorite features. However, developers and domain specialists speak different languages, have different priorities and widely disparate agendas. The developers want to cut scope down to the minimum that will be functionally adequate so they will have some chance of meeting the schedules imposed upon them; potential users want an application that will make their working lives as easy as possible.

The tension between these two forces inevitably brings an adversarial dynamic to requirements meetings that can be very entertaining. Domain experts can take the opportunity to express their resentment at the developer's intrusion into their domain and to laugh at the folly of the developer's attempts to capture the expertise and judgment acquired in a lifetime's professional endeavor in a few minutes of discussion. In turn, developers can mock the stakeholders for their lack of technical knowledge, their inability to express their know-how in a succinct and consistent manner, and to proclaim requests for even simple functionality as being impossible to implement for technical reasons that would take too long to go into.

Type #3: The Technical Meeting

MDD prescribes that all technical problems be solved "by committee". The basic method is:

1. Select a group of techies having maximum variation in technical opinion and preferences.
2. Put said techies together in a meeting room.
3. Direct them to reach consensus on the "best" solution to the technical problem.
4. Observe resultant fireworks and carnage.

MDD practitioners are not afraid to thrash out all technical issues amongst themselves, comparing the merits of varying approaches in an unstructured session of verbal sparring. As with many meeting-based outcomes, the determining factor is the relative rhetorical skill or obstinacy of the protagonists. Victory goes to whoever can best "ad lib" an argument to support their proposition, rather than whoever actually proposes the best solution.

Of course, there may not even be a "best" solution to the problem. It's likely there will only be a set of alternatives having different strengths and weakness. You'll find that if you let the fighting go on for long enough, eventually a compromise emerges that nobody is happy with, but which they will settle for simply for the sake of having the issue done with and getting out of the tense meeting room. This is how MDD forces issues to resolution – by escalating tension until it becomes unbearable.

From a technical lead's perspective, the MDD approach to design is also an excellent way to disguise your own incompetence. If you're in over your head in some technical arena, delegating all decisions to a meeting enables you to hide your lack of understanding and appear egalitarian at the same time. When the resulting design is implemented and found to be inadequate, the blame is spread amongst all the meeting participants rather than being focused upon yourself. It's a win-win situation for you.

The real magic of meetings is that they are like mini-corporations. Just as shareholders enjoy limited liability for the failure and misdeeds of the corporation, meeting participants enjoy a limited liability for the mistaken outcomes of the meeting. The meeting becomes an artificial entity unto itself; an additional, synthetic developer who is always willing to take the blame when something goes wrong.

The Progress Meeting

Progress meetings are at once the most uneventful and easiest to institute type of meeting. Their ostensible purpose is for team members to gather together and somehow collectively "update" their mutual awareness of the state of the project. Their real purposes are both symbolic and exculpatory. They provide an opportunity for the meeting organizer to give themselves the impression of active involvement with a project (even though they may see little of the team or its work at any other time), and also provide a way for the "hands off" manager to find out what is going on with their own project.

The most ineffective types of progress meetings are structured like this:

1. A chairman, usually the person who convened the meeting, reads through the action items from the previous progress meeting.
2. The assignee of each action item offers some excuse as to why they haven't attended to it, and then makes some vague resolution to do it before the next progress meeting.
3. The chairman reads out any new agenda items.
4. Each new agenda item is turned into a new action item and assigned to one of the meeting attendants, who promptly forgets about it.

5. The meeting is dismissed and the chairman writes up the minutes of the meeting and distributes them to the participants, who ignore them.

For most of the meeting then, there is only one-way communication from a speaker to a group of disinterested listeners. The same effect could be achieved through judicious use of a text-to-speech engine and Valium.

But there is great power hidden behind this apparently meaningless ritual. The chairman, in later distributing the minutes of the meeting, is in a position to engage in some historical revisionism. The minutes are supposed to detail the activities of the meeting and the decisions reached. But the one writing the minutes can generally write anything that they want, safe in the knowledge that hardly anyone will actually bother to read them. So if a decision doesn't go your way in the meeting, just change the way it is recorded in the minutes. You can even introduce items that were never discussed in the meeting, together with your preferred outcomes, safe in the knowledge that any participant who reads such an item but can't remember it from the meeting will probably conclude that they must have fallen asleep or been otherwise distracted during that part of the proceedings. Their unwillingness to admit their inattention means that your fabricated version of events will go unchallenged. The minutes are also invaluable for assigning blame when trouble occurs, as they can be used to substantiate claims that a particular resolution was arrived at with the agreement of all parties present (remembering that many will choose not to say anything at these meetings, lest they end up with work assigned to them, But their silence will forever condemn them to having offered implicit support for any decision you chose to put into the minutes).

Should the more rational members of the gathering ever object that these progress meetings seem pointless, you can always justify them by pointing out that they are an opportunity for communication to occur, and that communication is good. The complainant will be hard pressed to argue that communication is bad, and your point is won.

Review Meetings

Technical artifacts should always be reviewed by a group, for the practice offers numerous advantages ... to the reviewers, not the author of the work being reviewed. Reviews are a good opportunity to gang up on your enemies and humiliate them in front of an audience. Developers have a notoriously strong ego investment in their work, so tearing apart the finely tuned code they have been poring over for weeks is sure to provoke an interesting reaction. This is the principle goal of group code reviews. The reviewers function like a self-appointed council of inquisitors looking for evidence of witchcraft in the accused. And like a witchcraft trial, incriminating evidence can always be found, as few developers can write code or produce a design that cannot be criticized in some way for something. Review meetings also allow individuals to find fault with impunity, as any degree of pettiness or vindictiveness they might exhibit can be excused as a diligent attempt to make constructive criticism.

Once you can conduct all of the above types of meetings, and enter meeting mode at will, you may consider yourself a competent MDD practitioner.

Conclusion

So that's a brief overview of the magic that is Meeting Driven Development. This approach to software development has been around since the beginning of corporate activity in the programming arena. In many corporations, the developmental norm is indistinguishable from MDD. Meetings are so much a part of the corporate culture it would not occur to anyone to take any other approach.

You will find that many programmers are afraid of meetings, having come to view them as pointless, "busy work" activities. This is simply because they have not yet learnt to appreciate that futility is actually a strength of meetings, not a weakness. The ability to convincingly create the illusion of coordinated effort and activity is invaluable in many situations.

Meetings are not a knee-jerk reaction to problem solving as some suggest, but a vehicle for creating a synthetic corporate entity – a virtual member of the development team – that can adopt the responsibility for the participant's poor decision making and manifest inabilities. Only when they have abandoned their reflexive animosity towards meetings and recognized them for the ritual scapegoat that they are, can developers really appreciate the benefits of MDD.

---

[*] First published 30 Mar 2006 at http://www.hacknot.info/hacknot/action/showEntry?eid=84
[1] *Test Driven Development*, Kent Beck, Addison Wesley, 2003
[2] *Object Oriented Software Construction, 2nd Ed., Ch 11*, Bertrand Meyer, Prentice Hall, 1997

Extreme Programming and Agile Methods

Extreme Deprogramming*

In recent weeks I've read two books by cult survivors. The first, "Inside Out" by Alexandra Stein[1], describes her ten year embroilment in a Minneapolis political cult called "The O." The second, "Seductive Poison" by Deborah Layton[2], details the author's involvement with the "Peoples Temple," the religious cult lead by Jim Jones, who engineered the mass suicide of 900 of his followers in 1978.

Reading each I became aware of the similarities in the methods for control, manipulation and persuasion that both cults employed. It also occurred to me that those techniques were not just features of groups that would conform to the traditional definition of a cult, but also extended to what might be called benign cults. Think of the fierce loyalty of members of pyramid organizations such as Amway and Mary Kay; think of brands with a loyal consumer base like Apple and Harley Davidson[3]; and finally, think of the ardent supporters of Extreme Programming.

By examining some of the characteristic features of cults (benign and otherwise) and calling out their presence in the recently popular XP movement, I hope to throw some light on why this technical cult incites such fervor and emotion in certain members of the development community.

Drawing on the work of thought reform specialist Robert Lifton and others, consider the following characteristics of a cult, all of which are displayed by XP:

- Sense of higher purpose
- Loaded language
- Creation of an exclusive community
- Persuasive leadership
- Revisionism
- Aura of sacred science

Sense Of Higher Purpose

Cult members believe that they are privy to special truths and insights not known to the general community, and that it is their mission to spread this knowledge to others.

I could only laugh when I read Scott Ambler's response[4] to a letter taking issue with an article on outsourcing that he wrote for Software Development magazine. In the July 2003 issue he wrote "While it's nice that so many Indian companies have high CMM ratings, it doesn't reflect modern thinking about software development. CMM and Six Sigma have a tendency to lead to prescriptive, documentation-heavy processes." These are the words of a zealot, who is so convinced of the righteousness of his beliefs that he is willing to elevate them to the status of being representative of "modern thinking about software development." In unguarded moments, it is occasionally conceded that XP is not the answer to all software development problems, but that is certainly the attitude portrayed by many of its devotees. Spend any time reading `comp.software.extreme-programming` and you will not be able to help but notice the thinly veiled arrogance and elitist attitude behind the postings of many of XP's most zealous followers. This is definitely a group of people who think they have *got it*, and that anyone else not similarly enthused is a laggard.

Loaded Language

Cults create a custom vocabulary for their members. New words are invented, existing words are redefined, and a jargon of trite and pat clichés is developed.

Perhaps XP's most egregious effect on the broader software development community has been to infect communication with cutesy slogans and acronyms. No one could overlook the overuse the word "extreme" has been put to in the marketing of a host of unrelated products and concepts. The only common meaning amongst Extreme Programming, Extreme Project Management, Extreme Design and Extreme Testing is the implication of identifying a product that is sufficiently different from previous offerings to warrant purchase.

"Refactoring" has been abducted from its proper home in the algebraic texts and elevated to the status of an essential work method, which one must apply "ruthlessly." If we consider that "rework" or "restructuring" are essentially synonyms for "refactoring", we see that this piece of custom terminology is only dignifying the act of investing effort to correct ill-considered implementation decisions for no functional gain. In general usage, I have noticed the term being used as an even broader euphemism to disguise and minimize bug fixing and functional extension.

Particularly offensive is the frequent characterization of XP as "disciplined". XP may satisfy the weakest definitions of the word "disciplined" in so far as there is some regularity and control in its methods. But these minor concessions to true rigor are in fact just the leftovers remaining after the elimination of particular activities from a truly disciplined development process – one that includes formal documentation and design. The abandonment of these activities is precisely where XP's principal appeal to many lies – that there are fragments of a rigorous development process remaining after the unpleasant stuff has been cast aside is hardly sufficient basis upon which to claim that the overall work pattern exhibits discipline – unless one considers the determined pursuit of the path of least resistance to evidence discipline.

The XP jargon serves the same purpose as it does in any cult, to elevate the mundane to the significant through relabelling, and to misdirect attention away from failings and inconsistencies in the cult's value system. It is a shame that the XP community did not apply its own YAGNI (You Ain't Gonna Need It) principle to the invention of such novel terminology.

Creation Of An Exclusive Community

A cult provides a surrogate family for its members, who feel somehow separated and at odds with mainstream society.

Cults are a refuge for the uncertain. For those feeling lost or without direction, the faux certainty of a cult provides welcome relief. Software development is a field full of uncertainty. The increasing societal reliance upon software and the attendant but conflicting requirements for speedy and reliable development, has outpaced our ability to learn better ways to do our work. Faced with this unsatisfactory situation and desperate for a solution, the development community is vulnerable to the claims and promises made by XP. The fact that there is a community of enthusiastic proponents behind XP serves only to enhance its credibility via the principle of *social proof*[5]. In truth, the presence of such a community only evidences the widespread confusion about software development methods, coupled with the hope that there is some answer that doesn't entail unpleasant activities such as documentation.

Persuasive Leadership

Central to almost all cults is the founding member, a figure who through the strength of their own conviction is able to attract others to their cause.

The leaders of the XP movement are three members of the C3 project where XP was piloted – Kent Beck, Ron Jeffries and Ward Cunningham – and to a lesser extent the industry figures who have adopted it as their personal cause – Scott Ambler and Martin Fowler being amongst these. These people have generated an impressive amount of literature which forms the basis for the ever growing XP canon. They also serve as the XP community's ultimate arbiters of policy and direction. Reading the `comp.software.extreme-programming` newsgroup I notice people continually directing questions about their own interpretations of the XP doctrine to these central figures, seeking their approval and the authority of their advice. That there is a need for personal consultation in addition to the information provided by the large amount of literature on XP speaks of the imprecise and variable definition of the subtleties of XP practice. That knowledge of what is and isn't OK is seen to be held by a central authority and is not in the hands of the practitioners themselves, echoes the authoritarian distribution of sacred knowledge that is present in most cults.

Revisionism

Cults often craft alternative interpretations of world events, both present and historical, that serve to reinforce their belief system.

There are a number of examples of revisionism in XP. The most blatant concern the C3 project – the original breeding ground for XP. Proponents of XP repeatedly use this project as their poster child, the tacit claim being that its success is evidence of the validity of XP. However the reality is that the C3 project was a failure – ultimately being abandoned by the project sponsor and replaced with an off-the-shelf solution[6]. XP advocates have chosen to cast this failure as a success, by carefully defining the criteria for success that they claim is relevant. It is typical cult behavior to interpret real world events in a light that confirms existing beliefs, and to deny contrary evidence as being inauthentic.

One of the advantages of having a central authority is the ability to reconceive fundamental beliefs when necessary. The change in the attitude of the XP "inner circle" with regard to the production of documentation is an example of this. In its initial conception, documentation was regarded as unnecessary. In the light of real world experiences with XP, this stance softened to include the production of documentation "if you are required to." More recently, the philosophy has been stated as "if it's valuable to you, do it." Some would dismiss this as a result of XP's infancy, claiming that it is still being developed and refined; but I believe these shifts in position are the thought reformer's attempts to incorporate unflattering real world experience into their original ideation. Whatever real practitioner's experiences are, we can be sure that the primacy of XP doctrine will remain.

Aura Of Sacred Science

Which implies that the laws and tenets of the cult are beyond question.

Central to XP is the notion of the 12 core practices. These technical equivalents of the Ten Commandments are considered interdependent and so the removal of any one of them is likely to cause the collapse of the whole. This all-or-nothing thinking is typical of cults. Members must display total dedication to the cult and its objectives, or they are labeled impure and expelled from the community. This discourages members from questioning the cult's fundamental beliefs.

In the case of XP, the organizational circumstances required to perform all the core practices are so particular that it is doubtful if more than a handful of companies could ever host an authentic XP project. Therefore practitioners are forced to perform partial implementations of XP. If they are unsuccessful, then failure is attributed to the impurity of their implementation rather than any failing or infeasibility of XP itself. The quest for individual purity is a feature common to many cults, as is the contrivance of circumstances that render it ultimately unachievable.

Much is made of the "humanity" of the methodology, the transition from "journeyman" to "master", and the focus upon individual qualities and contributions. Consideration of these softer, cultural aspects of XP has devolved into the sort of pseudoscience we often find in new age cults centered on the notion of "personal power" and "personal growth". To quote one zealot "XP is a culture, not a method."[7] The elevation of a new and unproven methodology to the philosophical status of a Zen-like belief system demonstrates the skewed perspective that typifies cult mentality.

Conclusion

Whether you choose to label XP a cult is not as important as whether you recognize that it displays cult-like attributes. I believe that the psychological and social phenomenon underlying these six characteristics account in no small part for the current popularity that XP enjoys. I also believe that they point to its future.

Cults tend to have a very limited life. The hype and fervor can only sustain the devotion of the members for so long, and eventually they will look to other sources for inspiration – those leaving a cult are frequently drawn into another within a short time.

I believe that XP will eventually lose its luster and fall into disrepute like so many other religious, commercial and technical cults of the past. Many of the current adherents will cast about for a new cause to follow, and no doubt the marketing departments of the technical book publishers and software vendors will be only too happy to provide them with a new subject upon which to focus their devotion. Meanwhile, software projects will continue to fail or succeed with the same frequency as always, as our industry continues its search for a panacea to the ills of software development.

---

[*] First published 29 Jul 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=11
[1] *Inside Out,* Alexandria Stein, North Star Press, 2002
[2] *Seductive Poison,* Deborah Layton, Anchor Books, 1999
[3] *The Power of Cult Branding*, M. Ragas and B. Bueno, Prima Publishing, 2002
[4] *Software Development*, July 2003
[5] *Influence: The Psychology of Persuasion*, Robert Cialdini, Quill, 1993
[6] *Extreme Programming Refactored*, M. Stephens and D. Rosenberg, Apress, 2003
[7] *Enculturating Extreme Programmers*, David M. West

New Methodologies or New Age Methodologies?*

I first encountered the coincidence of the aesthetic and the technical in a secondary school mathematics class. After leading the class through an algebraic proof, my teacher said "You have to admit there's a certain beauty to that." As I recall, he was met by a room of blank stares, one of which was my own. I remember thinking "You sad, sad man." I really couldn't see how a mathematical proof could be called "beautiful". Beauty was an attribute reserved for the arts – a song could be beautiful, a painting could be beautiful, but a mathematical proof might at best be called "ingenious."

It wasn't until some years later at University, while studying data structures and algorithms that I would come to some appreciation of what my mathematics teacher had meant. An appreciation of certain algorithms would leave me with a smile on my face, and an ineffable feeling of satisfaction. I believe that to appreciate the beauty of something technical first requires the observer to care a lot about the subject at hand, and that what we experience has something to do with a sense of admiration for the mind that produced the thing, rather than the thing itself.

That it is possible to appreciate the technical in an aesthetic way is a realization that I suspect comes to many people after spending long enough in a particular technical field. But that aesthetic is a quality of an existing artifact, not a basis for its production. The sense of "rightness" that we associate with an elegant solution to a problem is the end result of a rather less romantic, technical struggle. It is not the starting point for that struggle, but rather a flag that indicates that we have arrived at a good resolution.

The New Age Methodologies

One of the more disturbing characteristics of the New Methodologies of software development is the tendency to impose a new aesthetic upon existing knowledge, and then interpret that aesthetic as evidence that something new has been discovered. Hence, we find the literature of the New Methodologies littered with references to Zen philosophy, craftsmanship, martial arts and personal empowerment. This is the stuff of pseudo-science and mysticism. By indulging in this sort of "discovery by metaphor," we risk descending into a stasis of vague, self-referential navel gazing that characterizes the delusional New Age movement.

In the following sections I look at a number of the software development metaphors that recent authors have proposed as a means of gaining insight into the software development process.

Personal Empowerment

The New Methodologies purport to be more focused on people than on process. This is often construed as empowering the programmers against a harsh and dictatorial management. The New Methodologies have values and principles at their foundation, on an equal footing with actual techniques and practices. Commonly touted values are communication, simplicity, feedback, courage and humility. No doubt these are worthwhile values, not only in software development but in practically every other field of human endeavor. So why would we chose to focus on these values particularly, and their relationship to software development? Perhaps the biggest effect of highlighting this arbitrary selection of values is to add a certain faux credibility to a methodology by associating it with noble concepts.

The irony of the "empowerment" message is that the vagueness of this values-based approach actually has the opposite effect – it disempowers the programmer. The power is placed instead in the hands of the methodologists, who must be consulted as to what the appropriate interpretation of these values is, in the situations the programmers actually encounter in the field. These spokesmen have become moral arbiters. A more precise and objective methodological foundation would empower individuals to unambiguously interpret the methodology's recommendations in their local environment, without the need to continuously seek clarification from the methodologists.

For more rational discussion of the predilections and working habits of software developers see:

- *"The Psychology of Computer Programming"* by Gerald Weinberg
- *"Peopleware"* by Tom DeMarco and Timothy Lister
- *"Constantine on Peopleware"* by Larry Constantine
- *"Understanding the Professional Programmer"* by Gerald Weinberg

Eastern Mysticism

Nowhere do the New Methodologies and the New Age movement intersect to more egregious effect than in the area of Zen philosophy. In an attempt to elevate the ordinary to the profound, or to disguise self-contradiction as sagacity, the New Methodologists will often invoke the inexplicable wisdom of Zen.

In the new edition of "Agile Software Development", Alistair Cockburn offers us this:

It is paradoxical, because it is not the case, and at the same time it is very much the case, that software development is mathematical ... engineering ... craft ... a mystical act of creation.

Worse yet, this obfuscating nonsense is later followed by:

The trouble with using engineering as a reference is that we, as a community, don't know what that means.

So the "engineering" metaphor is unacceptably difficult to understand, but koan-like homilies are OK?

Cockburn then introduces his Shu-Ha-Ri model of software development practice. Shu, Ha and Ri are the three levels of practice in Aikido, and roughly translate into *learn*, *detach* and *transcend*. In drawing this obtuse metaphor, Cockburn manages to simultaneously insult the intelligence of his readers and the martial arts tradition whose authenticity he is trying to co-opt. Much is made of the fact that software developers can be considered to pass through successive stages of facility that correspond to Shu, Ha and Ri. Nothing is made of the fact that the same analogy can be drawn with every other occupation whose practitioners grow in expertise over time.

One keeps waiting for the admission that all this armchair philosophizing is just self-deprecating jest, but it seems it is not going to be forthcoming. If you need a laugh, I'd encourage you read Kent Beck's message to a young extremist[1] and the comments that follow it. A greater pile of pseudo-intellectual backslapping you will not find anywhere outside of the self-congratulatory annals of the New Age movement.

Craftsmanship

The portrayal of "software development as craft" reached its most irksome zenith in Pete McBreen's loathsome book "Software Craftsmanship"[2]. The book presents a false dichotomy between engineering and craft. Engineering is mischaracterized as a soul-less and impersonal undertaking that ignores the contribution of, and variations between, individuals. However craftsmanship values the individual and nurtures their development through apprenticeship-like relationships with other practitioners.

McBreen makes the profound observation: "... large methodologies and formal structures don't write software; people do." Who'd have thought? I rather thought these structures were there to *support* the people in their efforts, not to supplant them. But apparently the Big M Methodologists are conspiring to eliminate the human contribution altogether and our only chance to save our jobs and our identities is to embrace our "craft" and our role in its development.

I'm sure many developers like to think of themselves as craftsmen – it strokes their egos and elevates their self-perceived status. However the notion of a craft is usually reserved for activities where artifacts are produced through manual skill and dexterity e.g. carpentry, painting, sculpture. In common usage you will also find it applied to certain intellectual artifacts (as in "well crafted prose") but not those artifacts of a more technical origin, of which software is surely one (we don't speak of "well crafted formulae")

To liken software developers to craftsmen may be superficially appealing, but it represents a retreat into the vague and inscrutable domain of the New Age theorist.

This Is Engineering

Engineering is the use of scientific knowledge to solve practical problems. It is characterized by activities such as planning and construction. Engineers maintain such values as precision, realism and integrity. Taking an engineering-based approach to software development in no way denies the significant influence that individual abilities and social dynamics exert over the outcomes we produce.

I believe engineering remains a suitable basis upon which we can make concrete advances in software development practices. The kind of New

Age humanism we are seeing incorporated into the New Methodologies only encourages endless philosophizing, metaphysical thinking and wasted effort spent in the exploration of non-falsifiable premises.

Follow The Money

If the New Methodologies continue to follow the examples of their New Age counterparts, it can only be a matter of time before they begin to employ some of the same merchandising tactics. Only half in jest, I contend that before too long we will see the following items available for your convenient online purchase:

- Tapes and CDs of lectures given by notable New Methodologists, that you can listen to in your car on the way to work. Titles may include "The Path To Agility" and "Power Programming".
- Office decorations in the mould of the Successories products. Inspirational plaques with panoramic landscapes and themes like courage, simplicity, humility etc. Matching mouse pads, mugs and badges.
- The "Agile Thought of the Day" email services
- Hokey accessories like diaries and calendars featuring slogans like "You Ain't Gonna Need It" and "Do The Simplest Thing That Could Possibly Work". The XP Programmer's cube[3] may be an early prototype.

Finally, let me leave you with a Zen parable. Make of it what you will:

Bazen and an Engineer were out walking together. Bazen turned to the Engineer and said, "Tell me Engineer, what is the sound of one hand clapping?" The Engineer, swatting at the air near one ear, replied "It's sort of a 'wooshing' noise, isn't it?" At this, Bazen was enlightened.

[*] First published 10 Nov 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=34
[1] http://c2.com/cgi/wiki?ToAyoungExtremist
[2] *Software Craftsmanship*, Pete McBreen, Addison Wesley, 2002
[3] http://xp123.com/xplor/xp0006/index.shtml

Rhetorical AntiPatterns in XP*

Over the past few years, I've spent more time in consideration of XP and its followers than is in the best interests of one's mental health. My pre-occupation with it springs from my broader interest in skepticism. It's fascinating to watch the same forces that drive cults, pseudo-science and other popular delusions at work in one's own profession. It's like driving past a road accident. It's tragic and disturbing, but so entrancing that you just can't look away.

One of the aspects of XP that is particularly intriguing is the way that certain rhetorical devices are used repeatedly to prop up the XP belief system in the face an uncooperative reality.

This post describes the four main rhetorical devices that XPers use to influence their audience and each other. Once you see how it's done, you'll find yourself able to "talk XP" like a native.

The four techniques are:

- Adopt A Tone Of Authority And Eschew Equivocation
- Make Bold Assertions And Broad Generalizations
- Use Evidence Whose Veracity Can Not Be Challenged
- Create Slogans And Neologisms

Adopt A Tone Of Authority And Eschew Equivocation

No matter what questions you might have, there is someone out there that is willing to sell you the answers. And although the vendors come in many different forms they have one characteristic in common – they all appear absolutely sincere and absolutely sure of themselves. So must you be if you are to talk like a true XPer.

Fortunately, the impression of authority is easily created with some linguistic sleight of hand:

- Never qualify your statements or concede error. If you say "I don't think that is true" nobody will notice. But if you say "That is absolutely false" you can capture people's interest and attention.

- Intimate that you are speaking on behalf of others. For example, the statement "Software developers don't work that way" is more compelling than the statement "I don't work that way." Stating that "Everybody knows X" is more impressive than stating "I know X."

Exercise some restraint with these techniques. It's easy to go too far and sound like a born-again prophet. You will find it useful to temper your pontifications with the occasional self-deprecatory statement, just to make it clear to your audience that although you know you are very wise, you don't think you're the Messiah.

Another way of elevating your own perceived authority is to denigrate others. For example, those not enamored of pair programming may be accused of being socially inept or sociopathic. More recently, we have seen attempts to attribute a distaste for pair programming to genetic disorders such as autism and Asperger's syndrome. Statements so personal are delightfully controversial, and can also be used to goad detractors into overly emotive responses, which can be interpreted as further evidence of mental instability. Applied frequently enough, such pathologizing will discourage your detractors from making public criticisms, knowing that they will be virtually waving their "freak flag" for all the world to see.

Finally, boost your own credibility by borrowing it from elsewhere. Make occasional references to:

- Eastern philosophies and spiritual traditions
- Movies, literature and personalities from pop culture
- Advanced mathematics and physics, particularly chaos theory and quantum mechanics
- Political ideologies

Make Bold Assertions And Broad Generalizations

XP rhetoric is characterized by broad and sweeping generalizations about software development practice, projects and developers. A classic example is the following, from Kent Beck:

Unacknowledged fear is the source of all software project failures.[1]

It takes a special kind of person to make such claims – specifically, one that is breathtakingly arrogant. If this arrogance doesn't come naturally to you, then you will have to affect it. The more spectacular and entertaining your statements, the better the chance that they will be turned into a sound bite or quoted by a journalist. The media loves attention grabbing one-liners and there is little you can say that is so ridiculous that the determined reader will not find some way to interpret it as both meaningful and insightful.

Do not let an absence of supporting evidence constrain your imagination. If detractors point out exceptions to your generalizations, simply dismiss those exceptions as being so atypical or statistically insignificant as to not warrant revision of an otherwise useful rule of thumb.

In argument, coupling these generalizations with baseless assertions is an effective "one-two" punch to your opponent's frontal lobes. If they should be rendered speechless at the audacity of your statements, seize the opportunity to change the subject or offer some non-sequitur, so that they will not have the opportunity to challenge you.

Most importantly, remember that the credibility of your propositions rests almost exclusively on your ability to deliver them with absolute conviction. The software development community are a gullible lot, and provided that you sound like you know what you're talking about, a great number of them will simply assume that you've got the facts to back it up. For those unencumbered by integrity, this is the ideal flock to lead out of the programmatic wilderness, if only you can make the cattle-call compelling enough.

To get you started, here are some bold assertions and baseless generalizations that are anti-XP in nature. Feel free to use them in your next exchange with an XPer.

- It is inevitable that XP will fade into technical obscurity, just like every other fad the software industry has witnessed in the last thirty years.

- The fervor with which XPers cling to their code-centric methodology betrays the underlying fear which drives them: the fear that if they should ever stop typing someone might realize that coding is their only skill. In a modern business context the ability to code is useless if not accompanied, in equal or greater measure, by the ability to perform a whole host of non-coding activities that XP does not even address.

- Extreme programming is not about programming. It is about the attempts of a small group of attention-seeking individuals to make their mark on the computing landscape.

- The irony of Extreme Programming is that to make it work in the real world, you have to moderate the "extremeness" to such an extent that you're left with just "programming."

Use Evidence Whose Veracity Can Not Be Challenged

The software development community has a very low evidentiary standard – somewhere approaching zero. In other words, personal observations and testimonials are the only corroboration that most will require for any statement you might make. Empirical software engineering is not a popular field and the task of gathering empirical data sounds altogether like too much hard work for most to be bothered with it. All the numbers and statistics that it generates make really boring reading. Additionally, it takes time to conduct experiments, and who has that sort of time when you're busy "riding the wave" of the latest technology fad?

These factors are a gift to you, the burgeoning XP orator. With suitably contrived "anecdotal evidence" you can justify any claim you might make, no matter how preposterous. Whether such evidence has any basis in fact is almost entirely irrelevant. Anecdotal evidence is qualitative in nature, which lends itself readily to exaggeration and confabulation. You can create anecdotal statistics, safe in the knowledge that nobody has any better information with which to challenge you. Here's an example from Robert Martin:

> *We find that only one in twenty programmers dislike pairing so much that they refuse to continue after trying it. About one in ten programmers start out being strongly resistant to the idea, but after trying for a couple of weeks about half of them find that pairing helps them.*[2]

If anyone *does* try to challenge your statistics, just ask them why they are so hung up on numbers, and suggest that an emphasis upon quantification in software development is unreasonable and impractical.

If the purported evidence originates from your own experiences, prefix it with "in my experience" and claim "I've seen it with my own eyes." Who could doubt that? If you want evidence to have come from someone else, to create the impression of independence, remember that you can always get the answers you want by asking the right questions of the right people.

Create Slogans And Neologisms

If you've ever wondered why the XP lexicon contains so many trite catch phrases like "embrace change" and cutesy terms like "planning game" and "YAGNI", then you've hit upon two of the most important features of the vernacular – slogans and neologisms.

Slogans are a frequently used marketing device. They're like the "hook" in a pop song – they are music to the ears of the masses. As an added bonus, they lend themselves to being parroted off dogmatically – which will discourage people from thinking (critically or otherwise) about the validity of the propositions they embody. XP slogans are the rhetorical equivalent of the pre-prepared meals that TV cooking show hosts introduce with the phrase "here's one I made earlier."

To get you started, here are a few anti-XP slogans you might like to put on a t-shirt or poster:

- Pair programming – for those with only half a brain
- eXtreme Propaganda not welcome here
- Embrace Change (You're Gonna Need It after you get fired)
- IfXPIsSoGreatWhyCan'tTheyFindTheSpaceBar?

Neologisms are a trademark of many methodologies. By creating new terms you also create the impression of invention; of having discovered or created something so novel that no existing term adequately describes it. Conveniently then, neologisms allow you to take old knowledge, give it a new name, and then portray it as being something new. What's more, if you created the term, then you have a monopoly over its definition, which you are free to change from time to time as suits your purpose. You can even furnish common terms like "success" and "simple" with methodology-specific definitions, if this is what it takes to preserve the truth of some rather brash statements you made earlier. Do not be hampered by the bug-bear of consistency. Feel free to develop conflicting definitions of terms, giving you the freedom to later invoke whatever definition is most convenient for the situation you're in. If anyone should highlight your self-contradiction, simply excuse it as evidence of a deeper wisdom that defies even *your* complete understanding.

A Catechism

To illustrate how these techniques can be used in combination, I offer you the following dialog that I may or may not have had recently (hey, it's anecdotal evidence – how are you going to challenge me?) with a hard-core XPer. I chose to abandon my usual skeptical mode of argument and get "down and dirty" with some XP lingo. I encourage you to try it sometime. It's quite liberating to be free of the constraints of logic, and the burden of proof.

XPer	Hey Ed, want to do some pair programming with me?
Ed:	No thanks - pair programming isn't for me.
XPer:	Have you tried it?
Ed:	Briefly, but I disliked it - which wasn't surprising. It's quite at odds with my personality.
XPer:	How long did you try it for?
Ed:	Oh - about four days or so
XPer:	(laughing) That's not nearly long enough. And you've got to make sure you're doing it right, otherwise it won't work.
Ed:	No … really. No amount of persistence is going to change the situation. I know enough about my own nature to say that with some confidence.
XPer:	But why not try it again? What are you afraid of?
Ed:	[switching to XP lingo] I'm afraid of ending up in a state of total cognitive surrender, like yourself and other similarly disillusioned XP zealots. Anyway – why do you need to program with someone else? Aren't you good enough to work by yourself?
XPer:	:[taken aback] It's not about "good enough", it's about "better". I'm more productive when I work with someone else.
Ed:	So you claim. If I claimed to be more productive with a whiskey and soda by my side, would that warrant charging up a bottle of Jack Daniels to the project? Playing around with novel work methods at the customer's expense is professionally irresponsible.
XPer:	But pair programming works! I've experienced it for myself!
Ed:	No, what you've experienced is having a nice time with a buddy. Then you justified it to yourself by claiming a productivity improvement. People see what they want to see.
XPer:	I don't think you can comment – you haven't really tried pair programming
Ed:	Or to put it another way – I'm not the slave to technical fashion that you are – which actually gives me a more objective viewpoint from which to comment. Pair programming is a fantasy - there is simply no evidence that it works. Those who think it does are kidding themselves.
XPer:	How can you say that? There was this university study that demonstrated experimentally that it works!
Ed:	Are you talking about the study by Laurie Williams at the University of Utah?
XPer:	Yeah – that's the one.
Ed:	Tell me – have you *read* William's thesis?

XPer: Well – no, but I've read *about* it.

Ed: So I can't comment on pair programming because I haven’t really tried it, but you *can* comment on experiments that you haven't even read

XPer: Look – I may not have read the details, but I know what it proved.

Ed: What it proved is that it's easy to do bad experiments, and that many software developers like yourself are gullible enough to believe anything they hear, so long as it fits in with their preconceptions. If you really knew about pair programming, you'd already know that the Williams experiment proves absolutely nothing.

XPer: I've paired with plenty of developers in the past, but nobody got upset about it like you. Have you got some kind of problem?

Ed: If you think that others should necessarily have the same preferences as you, then I’d suggest it’s *you* that’s got the problem. I'm happy for you to pair program if you want, but I must decline the offer to participate in your hallucination.

XPer: [shaking head] Ed, you've got to learn to "embrace change". The whole XP thing is taking off – "agile" is the way software development is gonna be from now on. Get on board or step aside.

Ed: "Change imposed is changed opposed."

XPer: How do you mean?

E: For one so agile, you’re a bit slow on the pick-up. In this context, it means that if you try and force people to work a way they don’t want to, then they'll fight back.

XPer I don't hear anyone fighting against XP.

Ed: Then where have you been for the last five minutes? You just demonstrated my point – people hear what they want to hear.

XPer: Ok, maybe some folks don’t get it, but there are plenty of people who do, and who are achieving success.

XPer: At least as many people have tried XP and failed. Some of them go on to claim success anyway, because admitting to failure would be too embarrassing. Most of them just say nothing and hope nobody notices their stuff-up. If you think the success-stories you read about in the media are representative, you’re kidding yourself. The real story is very, very different. "Success has many fathers, but failure is an orphan."

XPer: OK, maybe there's some truth to that. But you can’t be saying that all these XP proponents are lying?

Ed: No – not all of them, but some of them are, and some of them are exaggerating. The rest are probably what we call "pious frauds" - that is, they genuinely believe what they're saying, but are really misconstruing the influence of XP on their projects. It's easy to do if you play down the negatives and emphasize the positives.

XPer: Say – didn't you tell me once that you're a skeptic? Shouldn't a skeptic keep an open mind?

Ed: Yes, but not so open that their brains fall out.

---

[*] First published 19 Apr 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=51

[1] *Planning Extreme Programming*, Kent Beck and Martin Fowler, p8

[2] *Artima web logs forum*, posted November 15, 2003, R. Martin

The Deflowering of a Pair Programming Virgin*

In your readings of the voluminous XP canon, you will no doubt have encountered mention of the practice of Pair Programming[1]. If, like me, you are of a solitary disposition, you will have found yourself thinking – nice idea, but not for me.

Many of us are attracted to software development as a career because we enjoy the experience of solitary problem solving. We relish those times when we are "in the zone" – where our locus of concern narrows to exclude everything but ourselves, the keyboard and the problem at hand. This state can produce a feeling of mild euphoria, and gives us a place of retreat from the worries and concerns of our immediate environment.

The practice of Pair Programming puts an end to all of this. The problem solving medium moves from an interior dialogue to an exterior one. The silence we traditionally associate with deep thought and focused effort is replaced with the interaction and debate we more usually expect from a meeting or brainstorming session.

It was with some trepidation then that I recently accepted an offer from a colleague to engage in some Pair Programming as a way of extending my knowledge of certain subsystems of our application in which he had a greater degree of involvement than myself. The activity lasted about four days – long enough to complete the implementation and testing of a minor system feature in its entirety. The experience was an interesting one, but on the whole, not one that I'd care to repeat with any regularity.

Pair Programming studies so far conducted have tended to originate from academic environments, and so focus on novice-novice pairings amongst students. It is not clear that their findings translate into a commercial programming context staffed by more mature professionals. By contrast, myself and the colleague I paired with have been doing whatever it is that we do for 10+ years each. In the period described herein, we sat together for approximately six hours on each day, using the same person's computer each time.

Following is a point-form summary of my experiences over this period, both positive and negative.

Positives

- When pairing, one programmer keeps the other from goofing off and wasting time web surfing etc.
- You tend to be more diligent in the construction of unit tests and more careful in general when you know that someone is watching you and looking for error. Also, as a matter of professional pride, you don't want to be seen to be hacking by a colleague.
- The quality of code produced is marginally better than I would achieve at a first cut when coding individually.
- When two people have participated in the construction process, familiarity with the code is spread further amongst the team members which mitigates the dependence upon any individual. If there is no external documentation, it may be more efficient to acquire familiarity with a piece of code on this basis, than by the alternative – reverse engineering.
- There is the opportunity to pick up tricks and shortcuts from watching someone else go about the arcana of their job (e.g. learning to use IDE features that you were previously unaware of).
- Mistakes are picked up more quickly due to the overseeing of one's partner.

Negatives

- The constant interaction is very tiring. Most days I went home absolutely exhausted from the enervating effect of continuous dialog, and frequently with a headache.
- There is a lot of noise produced, which tends to disturb those in the surrounding area. A room full of pair programmers, as advocated by XP, would be very noisy indeed.
- There are numerous ergonomic problems when two people share a computer. My colleague prefers a conventional keyboard with international settings activated (he is bilingual), a trackball and a medium screen resolution. I prefer a split keyboard, no extended character set capability, a wheelie mouse and a slightly higher screen

resolution. We had to swap hardware whenever we "changed drivers," which was annoying. Had our preferences in screen resolution not been similar, working from the one VDU could have been impossible (for example, if one of us had low vision).

- There is a lot of "pair pressure" created from having someone watching every character you type. It tends to produce a self-consciousness that is inhibiting and constitutes a low-level and constant stressor.
- There is a tendency to feel constantly under time pressure when typing, because someone is waiting for your every keystroke. This produces a certain degree of "hurry up" sickness, which discourages any delay in doing more typing, such as that produced by thoughtful consideration of design issues.
- Groupthink can occur, even when there are only two people in the group. When you are working so closely with another, you are very wary of argument or disagreement, lest it sour the working relationship. Therefore people tend to agree too readily with one another, and seek compromise too quickly. Whoever chimes in first with a suggestion is likely to go unopposed.
- Time spent away from one's pair partner tends to be non-productive as your thoughts are dominated by the task the pair is currently tackling. This makes it difficult to effectively interleave other tasks with an extended Pair Programming session.
- Both myself and my colleague concede that we work in a different way when pairing than when working individually. Alone, our work patterns tends to consist of short bursts of productivity, separated by periods of mental slouching, by way of recuperation and cogitation. When pairing, those intermittent rest breaks are removed for fear of hindering someone else's progress, and because the low level details of different people's work habits will be unlikely to exactly coincide.

Conclusions

From this brief experience in Pair Programming it seems clear to me that the appeal (and therefore success) of the practice is likely to vary significantly between individuals. More gregarious programmers may enjoy the conversation and teaming effects, whereas more introverted programmers will find the constant interaction draining.

I am particularly interested to note that reports of Pair Programming experiences commonly available through the media tend to have a positive reporting bias. Experience reports of the form "we tried pair programming and we loved it" are not difficult to come by [2](which is not to say they are significant in number, but simply that a few studies are very frequently cited), but anecdotes that end "... and then he resigned because he couldn't bear the constant pair programming" are not as readily available. (for some of these, see the soon-to-be-reviewed-on-Hacknot "Extreme Programming Refactored: The Case Against XP").[3]

I don't believe my take on Pair Programming is likely to be singular. My personality type and communication preferences are not at all uncommon amongst developers. In Myers-Briggs terms I am an ISTJ[4], which is the most common personality type in the IT industry. I believe that many developers will find Pair Programming to be a difficult and ultimately unsustainable work practice – one that removes from their work day some of the basic elements that first attracted them to their occupation.

For a pairing of mature developers, I believe the effect on code quality is vastly overstated amongst the XP community. That there is some marginal improvement in the quality of the code when first cut seems clear. That this improvement justifies the investment of effort required to produce it, or that it could not be obtained more efficiently through regular code review techniques, is not at all clear.

Finally, I believe that Pair Programming is a very inefficient way to share knowledge amongst team members. The total man hours invested in doubling up can result in at best two people being familiar with the code being worked on. A good design document could guide an arbitrary number of future developers to an equivalently detailed understanding of the code, saving the expense of continual, unassisted reverse engineering on their parts.

Addendum

Shortly after posting this, a reader asked for the basis of my statement that ISTJ is the most common personality type in the IT industry. The findings of two large studies are relevant here, both of which I found referenced in "Professional Software Development", Steve McConnell, Addison Wesley, 2004, p63:

- *"Effective Project Teams: A Dilemma, a Model, a Solution,"* Rob Thomsett, American Programmer, July-August 1990, pp.25-35
- *"The DP Psyche,"* Michael L. Lyons, Datamation, August 15, 1985, pp. 103-109

McConnell cites these two studies as finding the most common personality type for software developers to be ISTJ. My statement generalizes this conclusion to the entire IT industry, which is obviously unwarranted.

McConnell cites further studies from Thomsett, Lyons, Bostrom and Kaiser as finding that ISTJs comprise 25-40 percent of all software developers.

[*] First published 16 Sep 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=22
[1] http://www.pairprogramming.com/
[2] http://www.cs.utah.edu/~lwilliam/Papers/ieeeSoftware.PDF
[3] http://www.hacknot.info/hacknot/action/showEntry?eid=23
[4] http://www.typelogic.com/istj.html

XP and ESP: The Truth is Out There![*]

"Eclipses occur, and savages are frightened. The medicine men wave wands – the sun is cured – they did it."– Charles Fort [1]

People have a vast capacity for self-deception. Even members of the scientific community, from whom we expect objectivity, can unwittingly allow their personal beliefs and preconceptions to color their interpretation of data. Professional ambition and wishful thinking can turn their stance from one of neutral observance into passionate adherence to a position, sustained by willful ignorance of contrary evidence. Such attitudes are common amongst the ranks of pseudo-scientists and paranormal researchers. Enthusiasts in this domain reward these ersatz scientists by buying their books and journals in numbers proportionate to the impressiveness of the alleged experimental findings. In doing so, they become complicit in their own deception.

Many of these enthusiasts labor under the misimpression that the existence of ESP, PK and other paranormal phenomena has been "proved" by creditable scientists. Many of the researchers are similarly deceived.

Curiously, we may be seeing exactly the same effects currently at work in the software development community with regard to XP. If there is sufficient desire to find "evidence" favorable to XP, it will be found. If there is sufficient reward for publication of XP success stories, they will be published. The belief that XP has been "proved" in the field can develop, if there is sufficient desire to believe it. And if sustaining that belief makes it necessary to ignore conflicting evidence and censor stories of failure, then that will also occur.

Be it XP trials or ESP experiments, there are two sorts of bias that make it possible to find significance where there is none, and sustain false belief. This post examines how these biases manifest in both domains.

Positive Outcome Bias: Embrace Change Or Exaggerate Chance?

Positive outcome bias is defined as:

The tendency of researchers and journals to publish research with positive outcomes much more frequently than research with negative outcomes.[2]

Suppose 100 researchers conduct an experiment in ESP. Each professor chooses a single subject who believes they have ESP and asks them to "sense" a series of randomly chosen Zener cards being "sent" to them by the person who selects the cards. Suppose that in 50% of these experiments, the subject achieves an accuracy greater than that which could be attributed to chance alone. The 50 researchers conducting those experiments are intrigued, and decide to conduct a further round of tests with the same subject. The other 50 researchers, knowing that failed attempts to detect ESP are unlikely to get them published, abandon their experiments.

In the next round of experiments, the same pattern occurs, and 25 more researchers give up. Eventually, all the researchers give up, but not before one has witnessed his subject beat chance in 6 or 7 consecutive experiments - which is quite a spectacular result! Deciding to neglect the final experiment that caused him to stop (figuring the subject was probably tired, anyway) the researcher writes up his results and sends them to the editor of the *Journal of Parapsychology*, in which they are published.

Consider the deception which results:

- The PSI research community's pro-ESP bias has been further confirmed by their receipt of this latest research evidence
- The readers of the *Journal of Parapsychology* are impressed with the evidence, and any pre-existing belief in ESP is further cemented.
- Other researchers, perhaps even some outside the PSI community, conclude "Maybe there's really something to this ESP stuff after all" and decide to conduct their own experiments in ESP, thereby propagating the effect into another round of investigations.

Note that neither the researcher who was published, the research community, nor any of the readers of the *Journal of Parapsychology* ever become aware of the 99 experiments that were abandoned because they were deemed unpublishable. Taken in isolation, the published result may be impressive. But taken in the context of the other 99 experiments that have silently failed, the published result may simply be an outlier whose occurrence was actually quite likely.

The following factors contribute to positive outcome bias:

1. Researchers who conduct uncontrolled experiments
2. Researchers who self-censor negative results
3. Researchers who can justify to themselves the imposition of optional starting and stopping conditions.
4. A publication environment that favors success stories

All three of these are features of the environment in which the software development community examines and reports on your favorite methodology and mine, XP:

1. XP is often trialed on a single project, on a non-comparative basis (controlled experimentation would be prohibitively expensive).
2. When an XP project fails, it will probably fail quietly. Companies and individuals have reputations to protect.
3. In a series of XP-related experiences, initial negative experiences are dismissed as "teething trouble". For an example, see Laurie William's pair programming experiment. Her dismissal of the last of four data sets, and devaluing of the first of those four data sets, is a good example of "optional starting and stopping conditions."
4. There can be no doubt that the IT media just loves those "XP saves the day" stories. Success stories sell magazines.

In such an environment, XP enthusiasts will declare "Wow, everywhere you look, XP is succeeding" – which is true. But it's in the places that you *haven't* looked that the real story lies.

Confirmation Bias

Confirmation bias is defined as:

The tendency to notice and to look for what confirms one's beliefs, and to ignore, not look for, or undervalue the relevance of what contradicts one's beliefs.

When it is pointed out to PSI researchers who claim to have successfully demonstrated ESP, that hundreds of non-PSI researchers have tried to replicate their results and failed, they sometimes attribute this to the ostensible influence that the attitude of both experimenter and subject can have over the results. An experimenter who is hostile towards the concept of ESP, they claim, can exert a negative influence over the results, thereby counteracting any positive ESP effects that may be present. This is one of the many "outs" PSI researchers have developed that enable them to attribute negative results to extraneous causes, and preserve only the data that is favorable to their preferred hypotheses.

We see exactly the same thing happening in the XP community's evaluation of experience reports from the field.

When presented with a claim of success using XP, the community accepts it without challenge, for it is a welcome confirmation of pre-existing beliefs. However, a claim that XP has failed is an unwelcome affront to their personal convictions. So these claims are scrutinized until an "out" is found - some extraneous factor to which the blame for failure can be assigned. If all else fails, one can claim, as PSI researchers are wont to do, that the attitude of the participants is to blame for the failure.

To illustrate, consider the tabulation below of the four types of experience reports that the XP community can be presented with. The columns represent the two basic modes of XP usage – full and partial. Either you're doing all the XP practices or you're only doing some of them. The rows represent the claimants assessment of the project outcome – success or failure. The table shows the interpretation an XP proponent can confer upon each type of experience report so as to *confirm* their pre-existing belief in XP.

	Full XP	**Subset of XP**
Success	*XP has succeeded!*	*See how powerful XP is? Even a subset of the practices can yield success!*
Failure	*You weren't doing xxx as well as you could have, or* *You weren't committed enough, or* *There's something wrong with you etc.*	*You weren't doing all the practices, so you weren't really doing XP.*

The XPers have all their bases covered. No matter what the experience report, there is no need to ever cast doubt upon XP itself – there are always rival causes to be blamed.[3] In this way, XP becomes non-falsifiable.

Conclusion

There is an "essential tension"[4] between being so skeptical of new technologies and methods that we miss the opportunity to exploit genuine innovations, and being so credulous that we are ourselves exploited by those willing to subjugate integrity to self-interest. Given the software industries' history of fads, trends and passing enthusiasms, we would be wise to approach claims of innovation with caution – where those claims are accompanied by fanaticism and zeal, doubly so. As Thomas Henry Huxley warned:

> *Trust a witness in all matters in which neither his self-interest, his passions, his prejudices, nor the love of the marvelous is strongly concerned. When they are involved, require corroborative evidence in exact proportion to the contravention of probability by the thing testified.*

There is no logical basis for dismissing out of hand every "next big thing" that comes along. But an awareness of confirmation bias, positive outcome bias and their contribution to the development of false beliefs should encourage us to seek evidence beyond that provided by popular media and effusive testimonial.

---

[*] First published 5 May 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=53
[1] Cited in *Voodoo Science*, Robert Park, Oxford, 2000
[2] *The Skeptic's Dictionary*, Robert Carroll, Wiley, 2003
[3] http://c2.com/cgi/wiki?IfXpIsntWorkingYoureNotDoingXp
[4] *Why People Believe Weird Things*, M. Shermer, Owl Books, 2002

Thought Leaders and Thought Followers*

Fowler On "Appeals To Authority"

For a brief, shining moment there was hope. Through the exaggeration and braggadocio that so permeates the conversation of the Agile community, there came a fleeting glimpse of self-awareness – a flash of social perspective that could have precipitated a greater moderation and rationality in the methodological discourse. And then it was gone – swept aside by the force of yet another ill-considered generalization.

I'm referring to a recent blog entry by Martin Fowler entitled *AppealToAuthority*.[1] In this entry, Fowler relates how he occasionally receives the comment "When a guru like you says something, lots of people will blindly do exactly what you say." Fowler denies the existence of such an effect, and counters that what appear to be appeals to authority may really be just an artifact of lazy argument or sloppy self-expression.

The argument from authority is everywhere in the Agile and XP communities, and is a far more potent force than Fowler seems to appreciate. Here are just a few ways that the various so-called "thought leaders" and "spokesmen" employ direct and indirect appeals to authority.

- Statements prefixed with "In my experience", combined with the suggestion that this experience is extensive, are attempts to cast the speaker as a seasoned veteran whose word should be taken seriously. Having many years of experience only establishes that one is old, not that one is correct.

- Sweeping statements and broad generalizations can make for powerful-sounding oratory, and suggest that the speaker possesses some kind of absolute knowledge i.e. that they are simply declaring information that they know to be factual. By abandoning the uncertainty and qualification, the speaker sacrifices accuracy for the sake of impact and elevates opinion to fact.

- By inventing and promulgating cute slogans, folksy homilies and other media-friendly sound bites, speakers encourage others to quote them verbatim and dogmatically. Such quotation invests the statement, and thereby the speaker, with a faux authority.

- With rare exception, the aforementioned comment from Fowler's being one such case, the "thought-leaders" and "spokesmen" rarely acknowledge, let alone reject, their decoration with such grand titles. There is no attempt to discourage the use of such titles, beyond the occasional token self-deprecation.
- Speakers claiming to represent the opinions and experiences of a group are naturally encouraging a view of themselves as leaders. Such speakers will not hesitate to claim "The Agile community believes X" or "The XP community does X", even though the communities in question have not been consulted or surveyed, and in fact may have wildly varying and inconsistent views on the matter.

Fowler's claim that appeals to authority are not a significant influence strikes me as disingenuous. Not only are such appeals frequent, they are at the very heart of the rhetoric. It should be kept firmly in mind that those most outspoken in this space are almost always consultants specializing in AM/XP.[2] Consultants make their money by promoting themselves as authorities on some subject, so that others will hire them for their perceived expertise.

Ruin Your Career With Agility

An interesting blog entry, author unknown, came to my attention recently. Entitled *How Agile Development Ruined My Career (Sort Of)*[3] it is the story of a Senior Director's attempts to introduce Agile work practices into a company, and the consequences for himself. I have commented on the blog itself, and the XP fraternity has just begun to dissect it on `comp.software.extreme-programming`[4] (posted 23 May 2004) which should make for entertaining reading.

[*] First published 24 May 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=55
[1] http://martinfowler.com/bliki/AppealToAuthority.html
[2] Agile Methods / Extreme Programming
[3] http://www.undefined.com/ia/archive/000158.html
[4] http://groups.google.com/groups?group=comp.software.extreme-programming

Requirements

Dude, Where's my Spacecraft?*

The Mars Polar Lander (MPL) that NASA launched in 1999 is now a rather attractive and very expensive field of tinsel-like shrapnel scattered over several square kilometers of the Martian surface. It is not functional in any capacity. It is no more. It has ceased to be.

Its demise was the result of the flight control software incorrectly answering the question that car-bound children have been plaguing their parents with for years – "are we *there* yet?" About 40 meters above the ground, the software succumbed to the constant nagging of its digital offspring and answered too hastily "Yes! We're there!" – triggering the shutdown of the MPL's descent engines. The craft's final moments were spent free falling towards the Martian soil at 50 mph (80km/h) – ten times the impact speed it was designed to withstand.

Monitoring the MPL's progress from Earth, NASA had expected a 12 minute period of broadcast silence during the descent to the landing area, due to the cant of the craft during re-entry. Shortly after touchdown, the MPL was scheduled to begin a 45 minute data transmission to Earth, but this transmission never occurred. NASA kept attempting contact with the MPL for the next six weeks, until finally giving up hope of ever hearing from it again.

Of course, it was not long before the faecal matter hit the rotary air distribution device.

In-depth mission reviews were conducted at NASA Headquarters, JPL and Lockheed Martin Astronautics. An independent assessment team was also established. Initially there were considered to be a number of possible causes for the mission's failure, but extensive investigations singled out one of them as being the most likely failure mode, with a high degree of confidence.

The assessment team concluded that a spurious signal from one or more of the touchdown sensors at the ends of the MPL's legs caused the software to conclude incorrectly that the craft had already made contact with the Martian soil and to therefore shutdown the descent engines prematurely.

However, this wasn't an unexpected hardware fault. The tendency of the Hall Effect touchdown sensors to generate a false momentary signal upon leg deployment was well known to NASA engineers, having been discovered in early testing. The software should have screened out these spurious signals, but this functionality was never actually implemented.

More precisely, the series of events leading to failure was likely the following:

1. 1500m above the surface of Mars, the legs of the MPL deployed. The touchdown sensor at the end of one or more of the legs generated a characteristic false touchdown signal while being deployed. The false touchdown event was registered by the flight control software and buffered.
2. 40m above the surface, the software began continuous sampling of the values from the touchdown sensors.
3. The first value read was the buffered false touchdown event that occurred upon leg deployment.
4. The software immediately triggered the shutdown of the Lander's descent engines, believing that the Lander was now on the surface of Mars.

Reasons For Failure

One of the main reasons the flight software did not behave correctly is because the definition of "correct" was changed in response to field testing. With respect to detecting touchdown, the system requirements initially stated:

"The touchdown sensors shall be sampled at 100 Hz rate. The sampling process shall be initiated prior to Lander entry to keep processor demand constant"

When the false signal characteristic of the touchdown sensors was later discovered, the following clause was added:

"However, the use of the touchdown sensor data shall not begin until 40 meters above the surface."

The intended effect of this addendum was to disregard the false touchdown signal previously generated during leg deployment at 1500m. This change was never propagated to the lower level software requirements.

Also note there is no explicit mention of the spurious signal generation. Even if this addendum had been propagated into the lower level requirements correctly, the software engineers would not have been aware that a false touchdown event might *already* have been registered at the time the use of the sensor data began.

Moral #1

The story contains two obvious lessons about requirements:

- Requirements tracking is useful in maintaining integrity between multiple requirements sources.
- Requirements should include a rationale i.e. specify *why*, not just *what.*

And now a few words from some XP spokesmen on requirements tracking:

I think I get, from the term, the idea of what RequirementsTracking is. It sounds like you keep track of changes to the requirements, who made the change, why they made it, when, stuff like that. If that's wrong, correct me now. If that's what RequirementsTracking is, I don't see the benefit. Please tell me a story where the moral is, "And that's why I am ever so happy that I tracked requirements changes." [1]

– Ron Jeffries, with assistance from Kent Beck

Moral #2

You would think that a thorough testing program would uncover the flight software's shortcomings. However, later testing did not detect the software's inability to cope with these signals because the touchdown sensors were incorrectly wired when the tests were performed. When the wiring error was discovered and corrected, the tests were not re-executed in their entirety. Specifically, the deployment of the Lander leg was not included in the test re-runs. The moral is: Thou shall *fully* regression test.

[*] First published 4 Nov 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=33
[1] http://c2.com/cgi/wiki?RequirementsTracking

User is a Four Letter Word*

The term "user" is not just a pronoun, it is a powerful buzzword that pervades the software development literature, to both good and bad effect. On the up side, the development community has been made aware of the dominating role that end user experience plays in determining the success or failure of many projects. On the down side, the message of the importance of user feedback to the development process has been adopted by some with uncritical fervor.

In their efforts to be "user focused," guided by simplistic notions of "usability," many managers and programmers uncritically accept whatever users tell them as a mandate. "The customer is always right" makes a nice slogan but a poor substitute for critical thought. If you want to deliver a product that is genuinely useful, it is important to moderate the user feedback you receive with your own knowledge of usability principles, and to seek independent confirmation of the information they relate. For it is a fact seldom acknowledged in the text books that users are frequently uninformed, mistaken or deliberately deceptive.

User Fraud

There are two types of fraud - the *deliberate* fraud and the *pious* fraud. Both make false statements; the former knowing that they are false, the latter believing them to be true. The user community contains both types.

Suppose you are writing a system that will facilitate the workflow of some subset of a company's employees. As future users of your software, you go to them to find out exactly how they do their work each day, so that you can understand their work processes. Some users find it difficult to articulate their basic work methods, even though they may have been in the same role for many years. Their routine becomes so internalized that it is no longer readily available by introspection. They may appear unsure and vague when describing how particular tasks are accomplished, and when you ask why things are done in a given way, you may get dismissive responses such as “Because that's the way we've always done it.”

Are you being told the truth? The naive developer will take what the user offers as gospel, and run away to implement it in software. The more experienced developer will simply take it on board for consideration, knowing that the user may be a fraud. Many users are pious frauds, in that they will give you their opinion on what workflow they and others are following, but state it as if it were an incontestable fact. Long-serving employees are very likely to consider themselves unassailable authorities on their company's processes.

But you must not lose sight of the fact that even the most genuine of users can be mistaken or have incomplete knowledge. When surveying employees who all participate in a common workflow, it is not at all uncommon to find that each participant has a different conception of the overall process. Sometimes there are only minor discrepancies between their individual accounts; sometimes there are direct conflicts and outright contradictions. This is particularly common in small organizations that function in a "cottage industry" manner, where nothing is written down and the work processes survive only through verbal instruction, not unlike the folkloric traditions that exist in tribes. The "Chinese whispers" effect can give rise to individuals having significantly different understandings of what is ostensibly a common work practice. Such users are not much to blame for their status as pious frauds, having become so through common psychosocial mechanisms.

Pious fraud also results from the common tendency to over-estimate one's own level of expertise in relation to others. For example, drivers involved in accidents or flunking a driving exam predict their performance on a reaction test less accurately than more accomplished drivers[1]. This *self-serving bias* will be also be present amongst your users, who may consider themselves experts in their domain and therefore convey their responses with greater authority and certainty than their true level of expertise actually justifies.

The user may describe a particular interface mechanism as having greater usability than another, when they are in fact only acknowledging the greater similarity of that design to the paper forms they are already familiar with. Users are not interface designers any more than drivers are automotive engineers.

On the border of pious and deliberate fraud are those users that are not lying outright, but neither are they making much effort to help you gather the information you need. They may simply be apathetic or cynical – perhaps having witnessed many failed IT initiatives within their organization in the past. When interviewed, their participation is begrudging, and they will make it obvious that they would rather be back at their post getting on with some "real work". They are only involved because their management has forced them to be so; they would really just like you to go away.

The answers you get from them may be the truth, but not necessarily the whole truth. Rather than describe to you all the variations and exceptional circumstances they encounter in the course of doing their job, they will simply give you a basic description of the usual way of doing things. Then it will be up to you to tease out of them all the boundary conditions and how they are handled. For the purposes of process automation, these special cases are particularly important.

Hardest for the software developer to deal with are the deliberate frauds. The developer is at a distinct disadvantage, for he is reliant upon the user for information, but is generally not familiar enough with the domain to be able to adduce that information's authenticity.

Asked to review documents that capture their workflow, the deliberate fraud may declare the document correct, when in fact they have not even read it. Or perhaps they actually *have* attempted to read it but are unwilling to admit that they have failed to understand it. A user may announce that their job requires judgments too complex or heuristic to be captured in software, when in fact they are simply unwilling to release their accumulated wisdom and expertise because they fear becoming expendable. The user may declare a particular procedure to be the correct one, but actually describe how they would *like* the procedure to be, in the hope that your software will result in things being done in accord with their personal preference.

Perhaps the most common ploy of the passive aggressive user is procrastination. When asked to participate in interviews or submit to any demand on their time, the user offers only perfunctory compliance, complaining that they just can't find the time to put in greater effort, given the demands of their existing duties. They know that if they demur frequently enough, you will probably stop assigning them tasks altogether.

Conclusion

There is a common tendency in the development community to conflate a "user focused" approach with one that unquestioningly accepts arbitrary dictation from users. The result is a gullible and over-confident development team that has unwittingly compromised their ability to effect the success of their own project.

While it is essential for developers to maintain a focus on their user's needs and expectations, they must be careful to think critically about the feedback they receive. To this end, it is important to independently verify the user's statements, obtain feedback from as broad a demographic as possible, and maintain an awareness of the potential for both deliberate and unintentional user error.

---

* First published 29 Jan 2006 at http://www.hacknot.info/hacknot/action/showEntry?eid=82
[1] *Incompetent And Unaware Of It*, J. Kruger and D. Dunning, Journal of Personality and Social Psychology, 1999, Vol. 77, No. 6, 1121-1134

Design

The Folly of Emergent Design*

One of the most pernicious ideas to proceed from the current focus on lightweight methodologies is that of Emergent Design. It's difficult to find a precise description of emergent design – most discussion on the subject carefully avoids committing to any particular definition. One of the most succinct descriptions I've encountered is this, from the *adaptionsoft* web site:

> *"Many systems eventually require drastic changes. You cannot anticipate them all, so stop trying to anticipate any of them. Code for today, and keep your code and your team agile."* [1]

Proponents of Emergent Design tout the following advantages of such an approach:

- Visible signs of progress appear more quickly.
- The system reaches a state in which it can be evaluated by customers sooner, which is useful for verifying existing requirements and teasing out as yet undiscovered requirements.
- The risk of "analysis paralysis" is eliminated.
- No effort is wasted in the preparation of infrastructure to support anticipated requirements that never actually manifest.
- An increased ability / willingness to adapt to changing requirements, as the development effort is not burdened by prior commitment to a particular solution approach.

Opponents of Emergent Design claim the following disadvantages:

- Exploration of alternative solutions takes much longer when using code as the vehicle for exploration, rather than a more abstract medium such as UML.
- The "code for today" approach discourages the reaping of long term savings in implementation effort by investing in supporting functionality in the short term.

Proponents will counter these by referencing the incremental nature of constant refactoring. Opponents will counter this with appeals to the benefits of a middle ground where "just enough" design is partnered with early prototyping [2]. Eventually, somebody makes comment on somebody else's mother and her preference for military footwear, and all hope of rational discussion is lost.

An Example Of The Hazards Of Emergent Design

As near as I can ascertain, the project upon which I am currently working employs Emergent Design, although there has been no explicit statement to that effect. At the beginning of the year there were one or two group design sessions, which identified the major subsystems of the product and how they would collaborate to achieve one of the principal use cases. Since then, any design efforts which have occurred have been of an incremental nature, and generally done "on the back of an envelope" as individuals have struggled to implement various aspects of a subsystem's functionality against pressing deadlines. Thus, developers have only done what was necessary to achieve the functionality need for the task at hand – which seems consistent with the philosophy of Emergent Design.

The resulting code base bears some interesting characteristics which I believe illustrate some of the difficulties inherent with the practical application of an Emergent Design approach. To illustrate, consider the following three classes from the application's current code base, presented here in abbreviated form:

```
public class YearLevel  {
  public YearLevel(NormYearLevel, Country, String, String);
  public getNormYearLevel() : NormYearLevel;
  public getCountry() : Country;
  public getScanText() : String;
  public getLabel(): String;
}

public class NormYearLevel {
  public static final NormYearLevel NORM_YEAR_1 =
    new NormYearLevel(1);
  public static final NormYearLevel NORM_YEAR_12 =
    new NormYearLevcel(12);
  private NormYearLevel(int aYearLevel);
}

public class RawYearLevel {
  public RawYearLevel(String aScanText);
}
```

The main purpose of this application is to process the responses of junior and secondary school students to multiple choice exams. A given exam may be taken by students from different countries and therefore different educational systems. The results are captured in individual and aggregate reports, which are printed and dispatched to the participating schools.

It takes as input the data files resulting from the optical scanning of the exam papers. Students indicate their "year level" as defined by the educational system in force in their country (a "year" is variously referred to as a "grade", "form" etc). For example, a student in year 3 in Australia would indicate a "3"; a student in Grade 4 in France would indicate a "4" and so on.

What is notable about these three classes is that they represent three different aspects of the same concept, and might well have been collapsed into a single abstraction. More significant than the choice and number of abstractions used to represent the concept, is the way these disparate representations came into being. Each was created by a different developer, working in a different subsystem from the others, and employing a philosophy consistent with Emergent Design. A review of the version control history for each class traces their genesis.

First came `RawYearLevel`, conceived of and implemented by a developer concerned with the early stages of the data processing pipeline, as a way of representing the student's literal indication of what year they were in.

In parallel with `RawYearLevel`, the `YearLevel` class was created by a second developer working in another subsystem, who was focusing on the opposite end of the pipeline, where the results are embodied in hard copy reports. The `YearLevel` class (without the `NormYearLevel` association) captured enough information to print on a report "This student was in Year 6" or "This student was in Grade 8", depending on the country and the educational system it employed.

Lastly came the `NormYearLevel` class, created by a third developer working in a subsystem between the two mentioned above, that was responsible for calculating individual and population statistics. In the course of these calculations it becomes necessary to relate a year level in one country with its educational equivalent in another country. So the concept of a Normative Year Level was introduced, and the country-specific `YearLevel` abstraction was augmented to be associated with it's normative equivalent.

Each of these classes has "emerged" from an individual developer's immediate need to implement some portion of a subsystem's functionality. To meet that need, they have done the simplest thing that could possibly work [3]. That often means writing a class from scratch. If another developer creates the same or a similar abstraction in parallel, each will be unaware of the duplication until their work is integrated. Sometimes it is considered simpler to get partial leverage from an existing abstraction. In either case, the imperative is to achieve the target functionality as quickly as possible, such is the time pressure the developers are under (a situation common to many development shops). It is by no means certain that the design issues surrounding these abstractions will *ever* be revisited.

Just Refactor It

The inefficiency of maintaining the above three abstractions is compounded by the amount of surrounding code that does little more than map from one type to another. Proponents of Emergent Design would suggest that the problem can be very simply overcome – just refactor the code. Of course, this is entirely possible. However there are some very real reasons why the abstractions have persisted in the application for 6 months or more, and have not been eliminated through refactoring.

- Nobody considers the refactoring to be of high enough priority to warrant spending our limited developer resources on. The task is not immediately related to any particular operational requirement, and so it is viewed as being less important than making functional progress.
- There is considerable psychological inertia associated with a body of code that is basically functional. Refactoring will mean losing that functionality for the duration of the refactoring task, and so superficially appears a retrograde step.
- The classes have become part of the vocabulary of the developers, and they have come to think of them as being an intrinsic part of the system i.e. their presence is not openly questioned.

Constraining Evolution Leads To Mutants

Emergent Design is frequently likened to the process of evolution. Proponents speak of "evolving a design" , the implication being that some software equivalent of natural selection is weeding out the inferior mutants, leaving only the fittest to survive. If this is the case, why have the three classes above not evolved into a better design? Or is that evolution yet to occur? Or are these three classes actually the fittest to survive already, for some suitable definition of "fittest"?

I conject that the practical application of Emergent Design so constrains the evolution of the design elements that we cannot expect such an approach to have a reasonable chance of giving rise to a good design.

Comparing the evolution of a software design with the evolution of a species, we see the following significant differences:

- Evolution can take its time exploring as many dead ends and genetic cul-de-sacs as it likes. There is no supervising authority standing by looking for visible signs of monotonic progress. There are no time constraints or fiscal limitations that require evolution to produce a workable result within a certain number of generations.
- Evolution can explore many alternatives in parallel, but a development group will rarely have sufficient resources to try a large number of different design alternatives in parallel. A very limited number of resources assigned to a design task must try alternatives in series, if at all. Obviously there is a strong tendency to stick with the first one tried that appears to hold promise.
- Evolution is objective in its evaluation of the success of each alternative. There is no attachment to a genetic alternative that is nearly good enough. However software developers often favor "pet" design approaches, or try and force non-optimal designs further than they should go because there is the promise of success just around the corner, and the attendant resolution of an uncompleted task. That is to say, it is very human to normalize deviation.
- Evolution is not required to be predictable. No one has bet their financial future on the lesser fairy penguin evolving heat dissipation mechanisms to cope with increasing Arctic temperatures, and doing it in no more than 3 generations. But stakeholders in software development efforts will commonly invest large sums to see successful designs produced (and thereby business problems solved) within a limited contract period.

You will find any number of elegant analogies in the Emergent Design literature – but finding one that addresses the above constraints is quite another matter.

For example, there is the delightful story (probably apocryphal) of the landscaping engineer who was asked to cement pathways at a University, after the buildings had been erected. Rather than predict the correct place to put the pathways, the engineer stood back for one semester and let the students make their own way between buildings. The furrows they wore in the ground were adopted as the courses for the cement pathways.

How very Zen... really, it's a terrific tale. I love it. But before we spin our prayer wheels and marvel at the engineers' wisdom, let's think of the liberties that the landscape engineer was allowed in pursuing such a solution method. Liberties which would be denied a great many University contractors in the real world:

- The landscape engineer was allowed to take the time necessary to wait for the paths to emerge. What if the University had required completion sooner than that – say, before the semester started?
- The landscape engineer was allowed an entire semester in which he was not required to demonstrate visible progress. What if a competitor had taken advantage of this lull and offered to complete the job using best guesses of the correct routes for the pathways.
- The landscape engineer was free to distribute the labor and materials cost over the course of the project as he saw fit. What if the budgeting system of the University had made allowances for expenditure on landscaping in this semester, but not in the following one?
- An entire University cohort spent a semester walking through the mud after every rainfall. They were willing to put up with this discomfort so that the engineer could let his design emerge. I wonder how the senior lecturers felt about this. More importantly, I wonder how those students in wheelchairs coped.

Emergent Design has the capacity to lead to some very elegant solutions – eventually. That design may be wonderfully efficient – if you have the financial stamina to await its arrival and the confidence that you will recognize it when it appears.

Conclusion

Does Emergent Design work? Of course - just look in the mirror. You and every other product of evolution is testament to the potential success of the approach.

Does that imply that it is a suitable model for designing software? No.

While the idea has aesthetic appeal, the practical context in which the emergence occurs makes all the difference. The requirements for timeliness and predictability in a software development project, together with the subjective nature of those who gauge the cost/benefit of a particular approach, mean that true, uninhibited evolution cannot occur. If the compromises embodied in an emergent design are consistent with our corporate priorities, then it will be by coincidence only – and that's too important a matter to leave to chance.

---

[*] First published 14 Oct 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=29
[1] http://www.adaptionsoft.com/xp_practices_simple_design.html
[2] Extreme Programming Refactored, M Stephens and D Rosenberg, Apress, 2003
[3] http://xp.c2.com/DoTheSimplestThingThatCouldPossiblyWork.html

The Top Ten Elements of Good Software Design*

"You know you've achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away." – Antoine de Saint-Exupery

Much is spoken of "good design" in the software world. It is what we all aim for when we start a project, and what we hope we still have when we walk away from the project. But how do we assess the "goodness" of a given design? Can we agree on what constitutes a good design, and if we can neither assess nor agree on the desirable qualities of a design, what hope have we of producing such a design?

It seems that many software developers feel that they can recognize a good design when they see or produce one, but have difficulty articulating the characteristics that design will have when completed. I asked three former colleagues – Tedious Soporific, Sparky and WillaWonga – for their "Top 10 Elements of Good Software Design". I combined these with my own ideas, then filtered and sorted them based upon personal preference and the prevailing wind direction, to produce the list you see below. A big thanks to the guys for taking the time to write up their ideas.

Below, for your edification and discussion, is our collective notion of the *Top 10 Elements of Good Software Design,* from least to most significant. That is, we believe that a good software design ...

10. Considers The Sophistication Of The Team Implementing It

Does it seem odd to consider the builder when deciding how to build? We would not challenge the notion that a developer's skill and experience has a profound effect on their work products, so why would we fail to consider their experience with the particular technologies and concepts our design exploits? Given fixed implementation resources, a good design doesn't place unfamiliar or unproven technologies in critical roles, where they become a likely point of failure.

Further, team size and their collocation (or otherwise) are considered. It would not be unusual for such a design's structure to reflect the high level structure of the team or organization that will implement it.

9. Uniformly Distributes Responsibility And Intelligence

Classes containing too much intelligence become both a point of contention for version control purposes, and a bottleneck for maintenance and development efforts. They also suggest that a class is capturing more than a single data abstraction.

8. Is Expressed In A Precise Design Language

The language of a design consists of the names of the entities within it, together with the names of the operations those entities perform. It is easier to understand a design expressed in precise and specific terms, as they provide a more accurate indication of the purpose of the entities and the way they cooperate to achieve the desired functionality. Look for the following features:

- The objective of the designed thing can be described in one or two sentences completely.
- The interface requirements of the entities are stated precisely.
- The contracts between an entity and its callers are stated precisely and contract adherence is enforced programmatically (Design by Contract).
- Entities are named with accurate and concrete terms, and specified fully enough to form a suitable basis for implementation.

7. Selects Appropriate Implementation Mechanisms

Certain mechanisms are problematic and more likely to produce difficulties at implementation time. A good design minimizes the use of such mechanisms. Examples are:

- Reflection and introspection
- Dynamic code generation
- Self-modifying code
- Extensive multi-threading

Sometimes the use of such mechanisms is unavoidable, but at other times a design choice can be made to sacrifice more complex, generic mechanisms for those easier to manage cognitively.

6. Is Robustly Documented

As long as a design lies hidden in the complexities of the code, so too does our ability arrive at an understanding of the code's structure as a whole. As the abstract structure becomes apparent to us, either through rigorous examination of the code or study of an accompanying design document, we gradually develop a course understanding of the code's topography. A good design document is used before or during implementation as a justification and guide, and after construction as a way for those new to the code base to get an overview of it more quickly than they can through reverse engineering. Captured in abstract form, we can discuss the pros and cons of different approaches and explore design alternatives more quickly than we can if we were instead manipulating a code-level representation of the design.

But as soon as the abstract and detailed records of a design part company, discrepancy between the two becomes all but inevitable. Therefore it is essential to document designs at a level of detail that is sufficiently abstract to make the document robust to changes in the code and not unnecessarily burdensome to keep up to date. A good design document should place an emphasis upon temporal and state relationships (dynamic behavior) rather than static structure, which can be more readily obtained from automated analysis of the source code. Such a document will also explain the rationale behind the principal design decisions.

5. Eliminates Duplication

Duplication is anathema to good design. We expect different instances of the same problem to have the same solution. To do otherwise introduces the unnecessary burden of understanding two different solutions where we need only understand one. There are also attendant integrity problems with maintaining consistency between the two differing solutions. Each design problem should be solved just once, and that same solution applied in a customized way to different instances of the target problem.

4. Is Internally Consistent And Unsurprising

We often use the term "intuitive" when describing a good user interface. The same quality applies to a good design. Something is "intuitive" if the way you expect (intuit) it to be is in accord with how it actually is. In a design context, this means using well-known and idiomatic solutions to common problems, resisting the urge to employ novelty for its own sake.

The philosophy is one of "same but different" – someone looking at your design will find familiar patterns and techniques, with a small amount of custom adaptation to the specific problem at hand. Additionally, we expect similar problems to be solved in similar ways in different parts of the system. A consistency of approach is achieved by employing common patterns, concepts, standards, libraries and tools.

3. Exhibits High Cohesion And Low Coupling

Our key mechanism for coping with complexity is abstraction – the reduction of detail in order to reduce the number of entities, and the number of associations between those entities, which must be simultaneously considered. In OO terms this means producing a design that decomposes a solution space into a half dozen or so discrete entities. Each entity should be readily comprehensible in isolation from the other design elements, to which end it should have a well defined and concisely stateable purpose. Each entity, be it a sub-system or class, can then be treated separately for purposes of development, testing and replacement. *Localization of data* and *separation of concerns* are principles which lead to a well decomposed design.

2. Is As Simple As Current And Foreseeable Constraints Will Allow

It is difficult to overstate the value of simplicity as a guiding design philosophy. Every undertaking regarding a design – be it implementation, modification or rationalization – begins with someone developing an understanding of that design. Both a detailed understanding of a particular focus area, and a broader understanding of the focus area's role in the overall system design, are necessary before these tasks can commence.

It is necessary to distinguish between accidental and essential complexity[1]. The essential complexity of a solution is that which is an unavoidable ramification of the complexity of the problem being solved. The accidental complexity of a solution is the additional complexity (beyond the essential complexity) that a solution exhibits by virtue of a particular design's approach to solving the problem. A good design minimizes accidental complexity, while handling essential complexity gracefully. Accidental complexity is often the result of the intellectual conceit of the designer, looking to show off their design "chops." Sometimes a "simple" approach is misinterpreted as being "simple-minded." On the other hand, we might make a design too simple to perform efficiently. This seems to be a rather rare occurrence in the field. As the scope of software development broadens at the enterprise level and attracts greater essential complexity, the reduction of accidental complexity becomes ever more important.

1. Provides The Necessary Functionality

The ultimate measure of a design's worth is whether its realization will be a product that satisfies the customer's requirements. Software development occurring in a business context must provide business value that justifies the cost of its construction. Also of significant importance is the design's ability to accommodate the inevitable modifications and extensions that follow on from changes in the business environment in which it operates.

But it is necessary to exercise great caution when predicting future requirements. An excessive focus upon anticipatory design can easily result in wasted effort resulting from faulty predictions, and encumber a design with unnecessary complexity resulting from generic provisions which are never exploited. Terms like "product line" and "framework" may be warning signs that the design is making high-risk assumptions about the future requirements it will be subject to.

It is easy to overlook the non-functional requirements (e.g. performance and deployment) incumbent upon the design. Taking different "views" of the design, in the manner of the "4+1" architectural views in RUP [2], can help provide confidence that there are no gaping holes (functional or otherwise) and that the design is complete.

---

[*] First published 18 May 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=54

[1] *The Mythical Man Month*, *Anniversary Edition*, F. Brooks, Addison-Wesley,, 1995

[2] *Rational Unified Process*, P Kruchten, Addison-Wesley, 1999

Documentation

Oral Documentation: Not Worth the Paper it's Written On*

The Agile Manifesto[1] states:

"The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."

Forgive me for questioning a holy proclamation, but isn't it rather well established that verbal communication is often incomplete and ambiguous, and that human memory is inaccurate and prone to confabulation? The plethora of psychological research in such areas as false memories, the veracity of eyewitness testimony, and the effect of predisposition on the interpretation of sensory data has surely given us a big hint that our perceptual and communicative capabilities are erratic and dubitable?

So where comes the apparently wide spread acceptance of (or at least, lack of challenge to) such outrageous Agile sophistry? For my part, it is difficult to ignore the manifest problems associated with a development team's reliance upon face-to-face communication. Over the last 3 or 4 months, as the inheritor of a code base whose authors preferred the "verbal tradition" style of documentation, I suffer daily from the flow-on effects of this laziness. Let me illustrate by providing you with a summary of a typical day for me in recent months, so you too can marvel at the feel-good richness and super-duper efficiency of face to face communication amongst software developers.

Fade in.

Scene 1 - a cubicle. Ed is slouched in an office chair staring forlornly at the screen in front of him. Except for the occasional insouciant jab at his keyboard, he gives the appearance of being comatose.

The day begins with my desire to extend the functionality of a legacy application, approximately 600K lines of code. I need to locate that portion of the code responsible for performing function X, so that I can insert function Y just after it. I go looking for function X amongst the code. I can't find it. In fact, I started looking for it sometime yesterday, and haven't found it yet. I check the folder marked "docs", to find it contains only a single `README.txt` file, the sole contents of which is the teaser "This directory will contain the docs" – apparently the dying message of a long extinct group of developers whose brains exploded before being able to make good on their promise. I find a piece of code that looks like it's in the same ballpark as the code I'm looking for, and examine the revision history of the file it is in, to find that it has principally been developed by "Bob". I must find Bob. I need to find Bob. Bob will know where function X is.

Here is my first problem. I cannot contact Bob directly, because I am but a lowly contractor. Bob is a valuable and in-demand member of my client's staff, and I can't just go up to him and steal his valuable time. There's a chain of command to be observed here! I must lodge a request with my manager to see Bob, who will forward that request to a liaison officer, who will forward that request to Bob's manager, who will then cue it up with Bob. If he's not too busy.

Scene 2 - a meeting room. Ed sits opposite a brown-skinned man wearing a turban.

The next day, I get to meet Bob. He can only spare 15 minutes to talk to me, because he's busy preparing for the next release of some whiz-bang new pile of crud. It's at this point that I discover that Bob's real name is "Sharmati Sanyuktananda", but everyone just calls him "Bob" for short. Bob is Indian. Bob's formal exposure to English was limited to the 15 minutes he spent reading "Miffy Learns English" while waiting in line at Immigration for his visa to be processed.

I try and talk with Bob, but it is like talking with Dr Seuss. At the end of 15 minutes, I have learnt almost nothing from him, and he keeps repeating something about public transport, which seems to have no relevance. His final word is "Sue", who I know is another member of the client's staff. So I contact my manager to organize some time with Sue.

Scene 3 - a meeting room. Ed sits opposite a nerdish looking woman wearing glasses with a very strong prescription.

Next day, I discover, to my significant relief, that Sue speaks English quite well. Unfortunately, her memory is a little hazy on the bit of code I'm asking her about. She remembers dealing with it about a year ago, but there's been a lot of water under the bridge since then. At this point, I am beginning to consider tying weights around my feet and jumping off that bridge. She can't tell me where functionality X is, but she's pretty sure it isn't where I'm looking. "Have you tried asking John?", she queries. So I contact my manager and request a meeting with another client staff member, John.

Scene 4 - a meeting room. Ed sits opposite a cool dude with sideburns and shoulder length hair.

Next day, John is disarmingly candid about the code I'm dealing with. "Oh yeah, I remember this crap", he begins. "We wrote that it in about a week, sometime last year, when we were up against the wall. It is absolute rubbish." "No kidding", I think. John is my guardian angel – he knows that function X got ripped out at the last moment, so they could meet their deadline. But then they put it back in a bit later, when things slowed down, and it's kept in a different module in the version control system. Which one? "You'll have to ask Declan", says John in a matter of fact way. I ask my manager to queue up some time with Declan.

Scene 5 - a cubicle. Ed is slouched in an office chair, browsing the advertisements on an employment web site.

My manager replies a few hours later, saying that Declan left the company a few months ago – maybe someone else knows. Have I tried asking Bob?

Fade to black.

And that, ladies and gentlemen, is the delight of face-to-face communication amongst software developers. See how efficient and effective it is? No one wasted any time writing nasty old documents, which saved them a bit of time – once. Everyone since then has wasted the time they saved, multiplied tenfold, trying to recover the information the original author could have documented in an hour or two, but was too busy, choosing to rely instead on good old "face to face" communication.

When it comes to the maintenance and extension of legacy code, and clearing the organizational hurdles associated with the handover of code from one party to another, a reliance on "face to face" communication is very convenient for the first generation of developers, and a chain around the leg of every other developer and organization involved thereafter.

It all sounds very folksy and appealing when you just say the words. If you're just talking in general terms about how much easier it is to have a bit of a chin wag with the bloke sitting next to you, then it sounds so reasonable to point out how much is being saved by just talking about stuff rather than writing it down. Of course! We'll just have a little chat about it and everything will be alright. That same simplicity is a large part of its appeal to many developers. Unfortunately, reality is not quite so simple.

For a maintenance programmer, the reality of dealing with your predecessor's reliance upon "oral documentation" is:

- The people you need to talk to are often not available – their time may be spoken for, or they may have left the company.
- The people that are available to talk with are often inarticulate techies with the verbal communication skills of a mime.
- The people you talk to have fuzzy memories, particularly where low level details are concerned. Frequently, they simply can't recall the information you need.
- The people you talk to all give you a different account of how things work. You're not getting the facts anymore, you're getting opinions and best guesses.
- The people you talk to have moved on to new duties and are not particularly interested in answering your queries about a system they would prefer to forget.

The "out" offered by XP/AM [2] and other idealistic retreats is that you just "do the documentation as needed". Brilliant! If only I'd thought of that, maybe I could've been a thought leader too! The problem is, "as needed" and "when time is available" are rarely coincident for reasons entirely beyond the developer's control. Try and convince a manager that you need to take a week out to catch up on some documentation. During that week you won't be writing code, you won't be making any functional progress towards a measurable or billable outcome, but the schedule will be taking a hit. Good luck with that one.

Fowler has a few delightful stories of "handover" scenarios in which face-to-face communication has been achieved by paradropping an "ambassador" into an enemy territory full of maintenance programmers, so that knowledge can be still be transferred verbally, and documentation produced as required by those maintenance programmers. I would like to enunciate a question that has long been in my mind, but heretofore unexpressed: "Martin, what part of the Twilight Zone do you live in, and where can I get a ticket?" Really ... is it just me or do the folksy anecdotes and one-off case studies that some Agile enthusiasts put forward sound just a little too contrived to be realistically transferred to your average corporate setting? Where are these companies they speak of, that have the latitude to abandon their normal procedures and protocols and set about bending over backwards in an effort to provide just the right climate to support these processes, no matter how involved the accommodation may be?

Whenever I read these fabulous accounts of the stunning success of AM/XP in some corporate environment, and how it didn't really matter that the team prepared no documentation whatsoever, I feel like I'm reading some sort of fairy tale, where everybody finishes their projects without difficulty, and then goes off to have a picnic in some bucolic setting, where they eat cucumber sandwiches and drink lashings of ginger beer. Hurrah!

By contrast, here's how handover happens in my world. One day – sometime before you've actually finished what you're working on – some pointy-haired manager comes up to you and says "You're changing to Project W tomorrow". No thought, no discussion, no campfire chat and singing of old spirituals. Just the immediate transferal of resources from one emergency to the next emergency. Whatever difficulties you might leave behind – too bad. What happens to the programmers that come after you is of no immediate concern. This dooms the poor sods to spending inordinate amounts of time, as I have recently, wandering the halls like a restless spirit, shuffling from one vague and apathetic source of information to the next.

The reliance upon face-to-face communication that the XP/AM contingent favor is not the straight-talking, light-weight, near-telepathic communicative fantasy of the Agile dream, but a prescription for pain and suffering for every maintenance programmer that has to come along and clean up after the original programming team has done a hit-and-run on the code base.

Are my experiences unique here, or do others find this whole "fireside chat" model of developer communication a little hard to swallow?

---

[*] First published 10 Jun 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=57

[1] http://www.agilemanifesto.org

[2] Extreme Programming / Agile Methods

FUDD: Fear, Uncertainty, Doubt and Design Documentation*

"Think twice, cut once" – Carpenter's adage

In the years that I've been doing software development, the one source of recurring dispute between myself and colleagues is the issue of design documentation. I am of the opinion that the production and review of design documentation significantly increases the chances of producing quality software, and that such documentation should be an integral part of the development of any piece of commercial software.

In the course of advancing this argument, I believe I have heard every counter-argument known to man (or "excuses," as I prefer to call them). It would require a small book to document them thoroughly, in all their variation and inventiveness, but the following list covers the main ones:

- We have a tight schedule and the sooner I begin coding, the better.
- The document will quickly drift out of synch with the code.
- I can always produce a design document later, if I have to.
- No one looks at design documents anyway.
- The information you capture can be obtained directly from the code.
- I'm paid to write software, not technical documents.
- The customer wants working software, not documents.
- Nobody does Big Design Up Front anymore.
- Never had to do it on any of my previous projects.
- Everyone on the team knows how the system is designed.
- A good design will emerge once we begin coding.
- It's better just to write the code, then recover the design later with a CASE tool.
- I comment the source code thoroughly.
- You can't really understand how the software will work until you write the code.

I'm not going to try and disprove any of these statements. The state of empirical research in the area and the vagueness of many of the statements themselves forbids disproof. Additionally, it is quite possible to develop and deliver software without a shred of design documentation. Indeed, it is common practice.

But I believe that we can do *better* with design documentation than without it. In other terms, though a tradesman might achieve his end with blunt tools, the going is harder and the result messier than if he had used sharp tools. My experience suggests that design documentation is a sharp tool that we blunt with our own misconceptions and false beliefs about the role it plays in the development process. Given that I can't prove that to you, I will try and persuade you of it by challenging some of the beliefs underlying the above statements.

It should first be acknowledged that for many developers, the notion of writing documentation of any type is a task they anticipate with the same distaste as root canal work. In other words, any of the above stated reasons for eschewing design documentation may really just be an attempt to rationalize the real reason:

I hate writing documentation

I believe the enmity toward documentation that we see so much of in the development community derives largely from the cognitive shortcomings (real or perceived) of the average software developer. Many developers come from mathematics, science and engineering backgrounds, and talent in those areas is often accompanied by a proportional lack of ability in the humanities. Documentation requires expression in natural language, and a disturbing number of developers have approximately the same facility with the written word as a high school junior. Nobody enjoys doing things that they're no good at. It's frustrating and tiring.

From the reasons given above, I have tried to distill the core underlying beliefs.

- Well written/commented code substitutes for design documentation
- The team already knows the design, so there's no need to document it
- Code is the only meaningful work product and sign of progress
- The maintenance cost of design documentation is prohibitively high

Let me challenge each of these beliefs in turn.

Well Written/Commented Code Substitutes For Design Documentation

Design documentation can provide value before the code is even written.

Senior technical staff frequently maintain an architecture-level view of the system being developed, leaving front-line developers to focus on whatever functional area they are currently preoccupied with. These are two distinctly different mindsets, and switching back and forth between them is tiring. When you've got your head buried in a complex multi-threading problem, you're not inclined to be thinking about how your code fits into the overall scheme of things. Similarly, when you're sorting out architectural issues, you're not concerned with lower level implementation details. By having the design of a low level subsystem reviewed by someone with a high level view of system structure, we can ensure that individual units of work go together in an architecturally consistent manner.

Additionally, the very act of externalizing a design to a level of detail that convinces a reviewer that it is sufficient, can lead the developer to discover aspects of the problem they might otherwise gloss over in their haste to begin coding. The problem with "back of the envelope" designs and hastily scribbled whiteboard designs is that they make it easy to overlook small but problematic details.

The Team Already Knows The Design, So There's No Need To Document It

Those who have taken part in the construction of a system have had the opportunity to witness the evolution of its design and absorb it in a piecemeal fashion over a period of time. But new team members and maintainers are thrown in at the deep end and confronted with the daunting task of gaining sufficient familiarity with an unknown body of code to enable them to fix and enhance it. For these developers, design documentation is a blessing. It enables them to quickly acquire an abstract understanding of a body of code, without having to tediously recover that information from the code itself. They can come up to speed with greater ease and more quickly than they might without the guidance of the design documentation.

Code Is The Only Meaningful Work Product And Sign Of Progress

This statement is true if the only lifecycle activity you recognize is coding, and the only goal towards which you proceed is "code complete." As a design matures and different aspects of the solution space are explored, the designers' understanding of the problem deepens. This progress in understanding is real progress towards a solution, even though it is not captured in code. The exploration and evaluation of design alternatives is also real progress, the end result of which is captured in a design document.

The Maintenance Cost Of Design Documentation Is Prohibitively High

Many developers view design documentation as a programmatic afterthought; something that you do after the real work of writing code is done, perhaps to satisfy a bureaucrat and create a paper trail. Any type of documentation produced in such a desultory fashion and out of a sense of obligation is likely to be low in quality, and of little use. So the preconception becomes a self-fulfilling prophecy.

It's not difficult at all to create useful design documentation, as long as you know what use you're going to put it to. I've found that useful design documentation can be achieved by following these two simple guidelines:

1. Include only those details that have explanatory power. There's no need to put every class on a class diagram, or to include every method and attribute. Only include the most significant classes, and only those features that are critical to the class's primary responsibilities; generally, these are the public features. Omit method arguments if you can get away with it. In other words, seek minimal sufficiency. This also makes the resulting document more robust to change.

2. Focus on dynamic behavior, not static structure. If possible, restrict yourself to a single class diagram per subsystem. Associations and inheritance hierarchies are relatively easy to recover from source code, but the interactions that occur in order to fulfill a subsystem's main responsibilities are much harder to identify from the code alone. This is why reverse engineering of interactivity diagrams by CASE tools is ubiquitously done poorly. The primary function of the design document is to explain how the classes interact in order to achieve the most important pieces of functionality

That code can be written in such a way as to obviate the need for documentation is a retort of the documentation-averse that I've been hearing for many years. Those not keen on commenting their code will appeal to the notion of "self-commenting code". Those not keen on design documentation will claim "the code is the design". This phrase, as it is commonly used, is intended to convey the idea that the code is the only manifestation/representation of the software's design that can be guaranteed to be accurate. While a design document will drift out of synch with the code, the code will always serve as the canonical representation of the design it embodies.

I believe such reasoning constitutes a scarecrow argument in that it presents an image of design documentation as necessarily so detailed and rigorous that it is fragile and brittle. Certainly it is possible to write design documentation in that manner, but it is also possible to make it quite robust by exercising some common sense regarding content and level of detail.

To the XPers who promote such fallacies, I would ask this:

> *"If you believe you can write code in such a way that the cost of change becomes negligible, why can't you employ those same techniques to write design documentation with the same properties? A design document does not demand the same accuracy or contain the same complexity as source code; so why can't you just refactor a design document with the same ease with which you refactor your code?"*

This inconsistency points to "the code is the design" argument as a failed attempt to rationalize personal preference. Twiddling with the code is fun, twiddling with diagrams is not (apparently).

Conclusion

Explicit consideration of design as a precursor to implementation has numerous benefits, most of which have their origin in the limited abilities of the brain to cope with complexity. Embarrassingly, there are those in our occupation who would deny the applicability of the mechanisms commonly employed in other fields to cope with these limitations. Abstraction, planning and forethought are as useful to software engineers as civil engineers. Design recovery from complex artifacts is just as difficult for us as for those in other construction-based occupations.

To get value from design documentation:

- Make it a part of your development cycle - don't treat it as an optional afterthought. Document as part of the design of each subsystem (NB: design documentation does not imply BDUF).
- Keep it as concise as possible, in the interests of maintainability.
- Eschew CASE tools offering round trip engineering and use a simple drawing tool (personally, I like the UML stencils in Visio).

- Concentrate on capturing dynamic behavior rather than static structure.

* First published 27 Jan 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=46

Programming

Get Your Filthy Tags Out of My Javadoc, Eugene*

Recently I've been instituting a code review process on a number of projects in my workplace. To kick start use of the process, I took a sample of the Java code written by each of my colleagues and reviewed it.

While doing so I was struck by the degree of individual variation in the use of Javadoc comments, and reminded of how easy it is to fulfill one's obligation to provide Javadoc without really thinking about how effectively one is actually communicating.

I think the quality of Javadoc commenting is important because - let's be honest - it's the only form of documentation that many systems will ever have.

Here are some of the problems in Javadoc usage that I frequently observe:

- Developers never actually run the Javadoc utility to generate HTML documentation, or do so with such irregularity they can have no confidence that their copy of the HTML documentation is up to date.
- Developers use their IDE's facility to auto-generate a comment skeleton from a method signature, but then fail to flesh out that skeleton.
- HTML tags are overused, severely impairing the readability of comments when viewed as plain text.
- Comment text is diluted with superfluous wording and duplication of information already conveyed by data types.
- Valuable details are omitted e.g. method pre-conditions and post-conditions, the types of elements in Collections and the range of valid values for arguments (in particular, whether an object reference can be `null`).
- The conventional single sentence summary at the beginning of a method header comment is omitted.
- Non-public class features are not commented.

My conclusion is that many developers are just "going through the motions" when writing Javadoc comments. With a little more thought, more effective use of both the author's and the reader's time can be made.

I propose the following guidelines for effective Javadoc commenting ...

Do Not Use HTML Tags

This maximizes the readability of the comment when viewed *in situ*, and saves the author some time (which is better spent adding meaningful text to the comment). Use simple typographic conventions[1] to create tables and lists.

Javadoc All Class Features, Regardless Of Scope

While third parties using your code as an API don't need it, the developers and maintainers of your code base do - and they are your principal audience.

Don't Prettify Comments

Cute formatting such as lining up the descriptions of `@param` tags wastes space you could devote to meaningful description and makes the comments harder to maintain.

Drop The Description For Dimple Accessors

For methods that simply set or get the value of a class attribute, this sentence duplicates the information contained in an @param or @return clause respectively.

Assume `Null` Is Not OK

Adopt the convention that object references can not be `null` unless otherwise stated. In the few circumstances where this is not true, specifically mention that `null` is OK, and explain what significance the `null` value has in that context.

Use Terse Language

Feel free to use phrases instead of full sentences, in the interest of brevity. Avoid superfluous references to the subject like "This class does ...", "Method to ...", "An integer that ...", "An abstract class which ...".

Be Precise

- For classes: precisely describe the object being modeled.
- For methods: describe the range of valid values for each `@param` and `@return`.
- For fields: describe the types of objects in Collections and the range of valid values.

[*] First published 6 Aug 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=14
[1] http://docutils.sourceforge.net/rst.html

Naming Classes: Do it Once and Do it Right*

The selection of good class names is critical to the maintainability of your application. They form the basic vocabulary in which developers speak and the language in which they describe the code's every activity. No wonder then that vague or misleading class names will quickly derail your best efforts to understand the code base.

Because we are called on to invent class names so frequently, there is a tendency to become somewhat lackadaisical in our approach. I hope the following guidelines will assist you in devising meaningful class names, and encourage you to invest the effort necessary to do so. As always, these are just guidelines and ultimately you should use your own discretion.

1. A Class Name Is Usually A Noun, Possibly Qualified.

The overwhelming majority of class names are nouns. Sometimes you use the noun by itself:

- `Image`
- `List`
- `Position`
- `File`
- `Exception`

Other times you qualify the noun with one or more words which help to specialize the noun:

Class Name	**Grammatical Breakdown**
`JPEGImage`	The noun `Image` is qualified by the noun `JPEG`
`LinkedList`	The noun `List` is qualified by the adjective `Linked`
`ParsePosition`	The noun `Position` is qualified by the verb `Parse`
`RandomAccessFile`	The noun `File` is qualified by the adjective `Random` and the verb `Access`
`FormException`	The noun `Exception` is qualified by the noun `Form`

When searching for a noun to serve as a class name, consider the following suffixes which are often used to form nouns from other words:[1]

Suffix	**Example Class Names**
-age	`Mileage, Usage`
-ation	`Annotation, Publication, Observation`
-er	`User, Broker, Listener, Observer, Adapter`
-or	`Decorator, Creditor, Author, Editor`
-ness	`Thickness, Brightness, Responsiveness`
-ant	`Participant, Entrant`
-ency	`Dependency, Frequency, Latency`
-ion	`Creation, Deletion, Expression, Enumeration`
-ity	`Plasticity, Mutability, Opacity`
-ing	`Tiling, Spacing, Formatting`
-al	`Dismissal, Removal, Committal`

2. Avoid Class Names That Have Non-Noun Interpretations

Suppose that while maintaining an application you come across a class called `Empty`. As a noun, instances of `Empty` might represent a state in which some vessel is devoid of contents. However the word "empty" can also function as a verb, being the act of removing all the contents of a vessel. So there is potential confusion as to whether the class models a state or an activity. This ambiguity would not arise if the class had been called `EmptyState` or `EmptyActivity`.

3. A Class Name Is Sometimes An Adjective.

There is a special type of class called a *structural property class*[2], which is often named with an adjective. Such classes exist to confer specific structural properties upon their subclasses (or implementers, in the case of interfaces). They are often suffixed with *-able*. Examples include:

- `Comparable`
- `Undoable`
- `Serializable`
- `Printable`
- `Drawable`

4. Use Commonly Accepted Domain Terminology

Specialist domains come ready-made with their own vernacular. This can be both a curse and a blessing. The down side is that newcomers to the domain have a lot of new terminology to master. The up side is that, once mastered, that terminology makes for efficient and precise communication with others fluent in the domain's jargon. Incorporating domain terminology in your class names is a good idea, as it succinctly communicates a lot of information to the reader. But you must be careful to use only terminology that is commonly known and has a precise definition, and ensure that your usage of the term is consistent with that definition. Avoid region-specific slang and colloquialisms. Examples:

- `DichotomousItem`
- `CorrigendaSection`
- `DeweyDecimalNumber`
- `AspectRatio`
- `OrganicCompound`

5. Use Design Pattern Names

Incorporating design pattern names like *Factory*, *Proxy* and *Singleton* into your class names is a good idea, for the same reasons that it is useful to use terminology from the application domain – because a lot of information is communicated succinctly. Just be careful not to get pattern-happy, and start thinking "everything is an instance of some pattern." Only refer to design pattern names if they have direct relevance to the intrinsic nature of the class. Examples:

- `ConnectionFactory`
- `ClientProxy`
- `AccountObserver`
- `DocumentBuilder`
- `TableDecorator`

6. Aim For Clarity Over Brevity

Many developers demonstrate a form of scarcity thinking when it comes to naming classes – as if there were a shortage of characters in the world and they should be conserved. The days when we needed to constrain identifiers to particular length restrictions are long gone. Today we should be focused upon selecting class names that communicate effectively, even if at the expense of a little extra length. With many developers using IDEs that support auto-completion, the traditional arguments in favor of abbreviation (typographical error and typing effort) are no longer applicable. The one case where abbreviation is warranted is specialist acronyms that are commonly used in the application domain e.g. `CMOSChip` is clearer than `ComplimentaryMetalOxide-SemiconductorChip`. Examples:

- `ProductionSchedule` is clearer than `ProdSched`
- `LaunchCommand` is clearer than `LaunchCmd`
- `ThirdParty` is clearer than `ThrdPrty`
- `ApplicationNumber` is clearer than `AppNum`
- `SystemCorrespondence` is clearer than `SysCorro`

7. Qualify Singular Nouns Rather Than Pluralize

When a class represents a collection of some type, it can be tempting to name it as the plural of the collected type e.g. a collection of `Part` classes might be called `Parts`. Although correct, you can communicate more about the nature of the collection by using qualifying nouns such as `Set`, `List`, `Iterator` and `Map`. Examples:

Class Name	**Group Semantics**
`PartList`	Parts are ordered
`PartSet`	Parts are unordered and each Part can not appear more than once
`PartPool`	Parts are interchangeable

8. Find Meaningful Alternatives To Generic Terms

Terms like `Item`, `Entry`, `Element`, `Component` *and* `Field` are very common and rather vague. If these terms really are the standard terminology in your application domain then you should use them. But if you are free to use class names of your own invention then search for something more specific and meaningful.

9. Imply Relationships With Other Classes

Naming a class provides you with the opportunity to communicate something about that class's relationship with other classes in the application. This will help other developers understand that class's place in a broader application context.

Some techniques that may be helpful in this regard:

- Use the name of a super-class or interface as a suffix e.g. call implementations of the `Task` interface `PrintTask`, `ExecuteTask` and `LayoutTask`.
- Prefix the name of abstract classes with the word `Abstract`.
- Name association classes by pre-pending and appending the class names on either side of the association e.g. the association between `Student` and `Test` could be called `StudentTakesTest`.

[*] First published 9 Mar 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=48

[1] *Bloomsbury Grammar Guide*, Gordon Jarvie

[2] *Object Oriented Software Construction, 2nd Edition*, Bertran Meyer

In Praise of Code Reviews*

I have a woeful sense of direction — the navigational abilities of a lemming combined with the homing instinct of a drunk. But like much of my gender, I continue to entertain the fantasy that I possess an instinctive ability to find my way, an evolutionary artifact of the male's traditional role as the hunter; an unerring inner compass that will guide me safely through the hunt of everyday life, despite voluminous evidence to the contrary. It is a fantasy that gets me in trouble on a regular basis.

Whenever I am driving to somewhere new, the scenario generally plays out like this: I begin the journey looking through my street directory, tracing out the path I need to follow. After memorizing the first few turns I set the directory down and depart, resolving to stop and consult the directory again once I've completed those turns. Within a few minutes I have traveled over the first part of the journey that I've already memorized, and have reached a decision point. Will I pull over to the side of the road and reacquire my bearings as planned, or will I just follow my nose? Invariably, I choose the latter.

"I'm bound to see a relevant sign before too much longer," I think. And so I drive on, keeping an eye out for the anticipated sign. If it doesn't shortly appear, I begin to make speculative turns based on my own "gut feeling" about which way to head. If I'm heading to a popular destination, I might simply follow the path I perceive most of the traffic is taking, figuring that they're all probably headed to the same place as I am. Through a combination of guess-work, dubious reasoning and random turns I eventually reach the point where I have to admit to myself that I'm lost. Only then will I pull over to the side of the road, get the street directory out of the glove compartment to find out where I am and how to get to my original destination from here.

This insane behavior has been a characteristic of my driving for many years. It usually manifests when I am driving home alone from some event which has left me feeling tired and distracted. I slip into a worn out fugue, adopt a "she'll be right" attitude and head off to goodness-knows-where. About a year ago, driving home from a job interview in a distant city, I strayed off course by over 100 kilometers – all the while resolutely refusing to pull over and consult my directory, which I could have done at any time.

Thanks to these unexpected excursions, I have seen parts of the country side that I might otherwise have missed, but I have no idea where they were or how to get back there.

So why do I do it? Why not spend five minutes by the side of the road working out where I've been and where I'm going, rather than just keep driving aimlessly in hope of finding some visible prompt to get me on course? As strange as the habit is, I think it's exactly the same behavior that many people exhibit when they make self-defeating decisions. It stems in part from short-term thinking.

Driving along in my pleasant reverie, I am faced with a choice. Stopping to consult my street directory will require some mental energy. I'll have to break the flow of my journey, find a significant landmark or intersection, locate it in the directory, and re-plot a path to my destination. The alternative is just to keep drifting along and hope for the best. If your scope of consideration is only the next few minutes, then it's very easy to decide to avoid the short-term inconvenience of pulling over in favor of continuing to do what you're already doing – even though it isn't working out and has already got you into difficulty.

A smoker indulges in similar thinking every time they light up. They know full well that they're killing themselves by having that next cigarette, but considering only the next five minutes, what is easier: Resisting the craving for a cigarette, or giving in?

This desire to minimize small, short-term pain even at the expense of significantly more pain in the long term is at the core of much self-defeating behavior.

We'll return to this theme in a moment. But first, a short divergence on code reviews.

Code Reviews

For many types of work it is standard practice to have one's work checked by another before the work product is put into service. Authors have editors; engineers have inspectors and so on. But in software development it is common for code to flow directly from the programmer's fingertips into the hands of the end users without ever having been seen by another pair of eyes.

This is despite there being a large body of empirical evidence establishing the effectiveness of code review techniques as a device for defect prevention. Since the early history of programming, a number of different techniques for reviewing code have been identified and assessed. A *code walkthrough* is any meeting in which two or more developers review a body of code for errors. A code walkthrough can find anywhere between 30 and 70 percent of the errors in a program[1]. *Code reading* is a more formal process in which printed copies of a body of code are distributed to two or more reviewers for independent review. Code reading has been found to detect about twice as many defects as testing[2]. Most formal of all is the *code inspection*, which is like a code walkthrough where participants play pre-defined roles such as *moderator*, *scribe* or *reviewer*. Participants receive training prior to the inspection. Code inspections are extremely effective, having been found to detect between 60 and 90 percent of defects[3]. Defect prevention leads to measurably shorter project schedules. For instance, code inspections have been found to give schedule savings of between 10 and 30 percent.

I estimate that about 25 percent of the projects I have worked on conducted code reviews, even though 100 percent of them were working against tight schedules. If we can save time and improve quality with code reviews, why weren't the other 75 percent of projects doing them?

I believe the answer is mostly psychological, and the basic mechanism is the same one that I engage in every time I go on one of my unplanned excursions in my car. The essential problems are short-term thinking, force of habit and hubris.

Suppose you have just finished coding a unit of work and are about to check it into your project's version control system. You're faced with a decision – should you have your code subjected to some review procedure, or should you just carry on to the next task? Thinking about just the next five minutes, which option is easier? On the one hand you'll have to organize the review, put up with criticism from the reviewers, and probably make modifications to your code based upon their responses. On the other hand, you can declare the task "finished', get the feeling of accomplishment that comes along with that, and be an apparent step closer to achieving your deadlines. So you make the decision which minimizes discomfort in the short term, the same way I decide to just keep on driving in search of a road sign rather than pull over and consult my street directory.

But then, you've got to rationalize this laziness to yourself in some way. So you reflect on previous experience and think "I've gotten away with not having my code reviewed in the past, so I'll almost certainly get away with it again". Similarly, I'm driving along thinking "I've never failed to eventually get where I'm going in the past, so I'll almost certainly get there this time as well." Complacency breeds complacency.

Finally, although it is difficult to admit, there is some comfort in not having your code reviewed by others. We would like to think that we can write good code all by ourselves, without the help of others, so avoiding code reviews enables us to avoid confronting our own weaknesses. In the same way, by following my nose rather than following my street directory, I can avoid having to confront the geographically exact evidence of my hopeless sense of direction that it will provide. Ignorance is bliss.

Even when you quote the empirical evidence to programmers, many will still find a way to excuse themselves from performing code reviews, by assuming that the touted reductions in schedule and improvements in quality were derived through experimentation upon lesser developers than themselves. The thinking goes something like "Sure, code reviews might catch a large percentage of the defects in the average programmer's work, but I'm way above average, don't write as many defects, and so won't get the same return on investment that others might." Unfortunately it is very difficult to tell simply by introspection whether you really are an above average programmer, or whether you just *think* you are. Most people consider that they are "above average" in ability with respect to a given skill, even though they have little or no evidence to support that view. For example, most of us consider ourselves "better than average drivers". The effect is sometimes referred to as *self-serving bias* or simply the *above average effect.*

Those that have bought into the Agile propaganda (can we call it "agile-prop"?) may have been deceived into thinking that pair programming is a substitute for code reviews. To the best of my knowledge, there is no credible empirical evidence that this is the case. In fact, there are good reasons to be highly skeptical of any such assertions – in particular, that a pair programmer does not have the independent view of the code that a reviewer uninvolved with its production can have. Much of the benefit of reviews comes from the reviewers different psychological perspective on the product under review, the fact that they have no ego investment in it, and that they have not gone through the same (potentially erroneous) thought processes that the original author/s have done in writing it. A pair programmer is not so divorced from the work product or the process by which it was generated, and so one would expect a corresponding decrease in ability to detect faults.

So we sustain self-defeating work practices the same way we sustain many other sorts of self-defeating behavior – by lying to ourselves and putting long term considerations aside.

Do Code Reviews Have A Bad Reputation?

There is perhaps another factor contributing to a hesitance to perform code reviews, which is the reputation they have as being confrontational and ego-bruising experiences. This reputation probably springs from consideration of the more formal review processes such as code inspections, in which the reviewing parties can be perceived as "ganging up" on the solitary author of the code, subjecting them to a famously unexpected Spanish Inquisition.

This is a legitimate concern, and it is certainly easy for a review of code to turn into a review of the coder, if a distinct separation is not encouraged and enforced. I therefore recommend that code reviews be conducted by individual reviewers in the absence of the code's author. This tends to depersonalize the process somewhat, and remove some of the intimidatory effect that a group process can have. There is in fact some evidence to suggest that an individual reviewer is no less effective than a group of reviewers in detecting faults in code.

The code can be printed out and written comments attached to it, or comments can be made in the source file itself, perhaps as "TODO" items that can be automatically flagged by an IDE. Personally, I prefer paper-based reviews because a paper-based review system is quick and easy to institute, and equally applicable to reviews of written artifacts such as design and requirements documents.

Conclusion

There is much to recommend the practice of conducting code reviews on a regular basis, and few negatives associated with them, provided they are conducted sensitively and with regard for the feelings of the code's author. All it takes is for one other programmer on your team to be willing to undertake the task, and you can establish a simple code review process that will likely produce noticeable benefits in improved code quality and reduced defect counts. Not everyone is good at reviewing code, so if you have the option, have your code reviewed by someone who demonstrates an eye for detail and is known for their thoroughness. If you have the authority to do so, it is well worth incorporating code reviews into your team's development practice, perhaps as a mandatory activity to be undertaken before new code is committed to the code base, or perhaps on a random basis. It may also be worthwhile to have junior staff review the code written by their more experienced counterparts, as a way of spreading knowledge of good coding techniques and habits.

When introducing code reviews, you will likely encounter some initial resistance, simply because the short-term thinking which has so far justified their absence is a habit that is superficially attractive and requiring of a certain determination to break. However, once they have had the opportunity to participate in code reviews, many programmers will concede that it is a habit worth forming.

---

[*] First published 27 Feb 2006 at http://www.hacknot.info/hacknot/action/showEntry?eid=83

[1] *Rapid Development*, Steve McConnell, pg 70, citing Myers 1979, Boehm 1987b, Yourdon 1989b

[2] *Ibid*, pg 71, citing Card 1987

[3] Ibid, pg 71

User Interfaces

Web Accessibility for the Apathetic*

If you're like me, you approach the subject of accessibility with a certain self-conscious guilt. On the one hand, you recognize that there are excellent ethical and legal reasons for making your applications – be they web-based or rich client – accessible to those with sensory or cognitive impairments; but on the other hand you can't ignore the fact that the extra work required to add that accessibility is only going to make a difference to a very small percentage of your users.

In recent years, the legal impetus has begun to gain strength, forcing those of us to action who might otherwise have been willing to put our internal ethics department on hold in the name of conserving time and energy. Having spent some time recently working inside a department of the Australian government, I have learnt that the issue of accessibility, in particular web accessibility, has a reasonably high profile. Because government web sites are required to adhere to accessibility guidelines[1], there has developed a group, comprised of either moralists or opportunists, who spend their time scouring the web pages of government web sites looking for non-conformances to use as the basis for legal prosecution. American courts have recently ruled that the accessibility requirements pertinent to US governmental web sites are also applicable to privately held web sites. Even your blog counts as material that is made "publicly available," and must therefore be equally available to all.

With these ideas in mind, and also to assuage my growing feelings of guilt regarding the accessibility (or lack thereof) of this site, I decided to undertake a bit of a site revamp, the cosmetic results of which you will already have noticed. This article provides a brief overview of the process I followed, and thereby gives a general introduction to the tools and techniques necessary to retro-fit accessibility to a site that was designed without specific consideration of that issue.

General Approach

In general, web accessibility can be achieved by adhering to the following two principles:

- Separate presentation from content by restricting your use of HTML to the standard structural elements, and using CSS (Cascading Style Sheets) to control the way that structure is presented.
- Emphasize textual content. Where non-textual content is used, always provide a textual equivalent.

A good portion of the details appearing below are in support of these two principles. The steps below show you how to transform a non-accessible web page into an accessible one.

Step 1: Ensure All HTML Elements Are Structural

Structural elements those which describe the semantic units of an HTML document. Examples of structural HTML elements are:

- `<h1> … <h6>`
- `<p>`
- `<ol>`
- `<ul>`
- `<img>`
- `<li>`
- `<div>`
- `<span>`

Over the years, browser vendors have added proprietary non-structural elements and attributes to the HTML grammar their browser understands, in an effort to differentiate their product from their competitor's. The result is a tag set which invites misuse, is interpreted differently (or not at all) in different browsers, and awkwardly combines content and presentation. By removing elements that specify some aspect of the document's presentation, accessibility can be improved.

Examples of non-structural HTML elements you should remove are:

- `<hr>`
- `<i>`
- `<b>`
- `<u>`
- `<big>`
- `<small>`
- `<font>`
- `<basefont>`
- `<br>`
- `<font>`

- `<tt>`

The layout effects produced by these non-structural tags can, and should be, achieved with style sheets. Using these tags only pollutes your HTML document with presentation information that may well be useless or misleading to those with low vision. For instance, `<b>` elements should be removed because bold text has no meaning to a blind user. This doesn't mean that text can't be made bold, but rather that CSS rather than HTML should be the means by which the bolding is achieved.

Note that in some cases there is a structural tag that you should put in place of the deleted non-structural tag. For example:

- If you have removed `<b>` tags that were used to emphasize words, insert `<strong>` tags where the `<b>` tags used to be.
- If you have removed `<b>` tags that were used to create a heading, insert a heading tag like `<h3>` where the `<b>` tags used to be.
- If you have removed `<b>` tags that were used to add emphasis, insert `<em>` tags where the `<b>` tags used to be.

In other cases, there is no structural element already defined in HTML that adequately captures a structural aspect of your web page, so you must invent your own using the `<span>` or `<div>` elements. For instance, you might create a of class "footnote" to denote footnote references:

```
<span class=”footnote”>This is a footnote.</span>
```

The way that span elements of class footnote are displayed is later specified in a CSS.

Step 2: Ensure All HTML Attributes Are Structural

Non-structural attributes should be removed for the same reasons that structural elements should be removed. Examples of non-structural attributes you can delete are:

- `align`
- `link`
- `alink`
- `halign`
- `valign`
- `background`
- `color`

- `text`
- `bgcolor`
- `vspace`
- `height`
- `width`
- `hspace`
- `border`

Again, all the layout effects that were produced by these attributes can be achieved with CSS, leaving the basic HTML document more accessible.

Step 3: Remove Misused Structural HTML Elements

Structural elements should not be used as ersatz layout mechanisms as this will confuse those accessing your web page with a text browser.

Examples of the misuse of structural elements for layout purposes include:

- Using empty paragraphs (`<p>`) to put a vertical space between consecutive blocks of text.
- Using the `<table>` element to achieve columnar alignment of material that is not inherently tabular.
- Drawing lines by stretching a 1-pixel `<img>`.
- Using `<blockquote>` purely to achieve indentation.

Step 4: Ensure All Non-Textual Content Has A Textual Equivalent

Users with visual impairment may use a text browser, Braille bar or screen reader to access your web page. These mechanisms can only deal with text as input. So you need to supply a textual equivalent to any non-textual content on your web page. A common examples is using the `alt` attribute of `<img>` tags to describe the significance of the image.

There are certain mechanisms which should be used sparingly, if at all, because they are inherently inaccessible. These include:

- Image maps
- Javascript
- Side-by-side frames

- Secondary windows
- Shockwave animations

Not only are these mechanisms difficult for some users to access, but they may be deliberately disabled by any user in their browser.

Step 5: Add In Attributes Or Elements That Aid Accessibility

There are a few HTML structural elements and attributes that are particularly helpful from an accessibility perspective:

- The `<abbrev>` and `<acronym>` elements can be used to specify the expansion of abbreviations and acronyms when they first occur in a document.
- The `<th>` element should be used to identify column headers. Tables are linearized in text browsers, and knowing which table cells are headers helps the user interpret them.
- In HTML forms, use the `<label>` element around the form labels. Additionally, field labels should be immediately to the left of, or immediately above, the field.
- Provide a logical tab order for elements by specifying the `tabindex` attribute for `<input>` elements.
- Use the title attribute of `<a>` elements to provide more information about the target of the hyperlink.

Checkpoint

At this point, you should have an HTML document that is marked up solely with structural elements and attributes. This is a good time to preview your page in a text browser like Lynx[2], or with a screen reader like IBM Home Page Reader[3].

It is also a good time to run your HTML through one of the automated accessibility-checker sites on the web. Such sites enable you to provide your HTML – either directly with cut/paste, or by nominating a URL – and then scan the document looking for accessibility problems. I found `www.bobby.com` to be quite useful.

Step 6:
Recreate The Layout Using Cantankerous Style Sheets

And now for the tricky bit. Converting your web page to use only structural HTML elements and attributes is easy compared to using CSS to achieve your desired layout. Mostly the difficulty stems from the variations in the way different browsers render CSS directives. Behavior of "floating" elements seems to be particularly problematic. Therefore it is essential to test the layout in as many different browsers as you can. This lack of standardization in behavior is the most frustrating aspect of using CSS. I found the following books useful in getting up to speed on CSS:

- *CSS - The Definitive Guide 2nd Edition*, Eric A. Meyer, O'Reilly Media Inc, 2004
- *CSS - Designing for the Web 2nd Edition*, H. Lie and Bert Bos, Addison Wesley, 1999
- *More Eric Meyer on CSS*, Eric Meyer, New Riders, 2004

Once you've got a style sheet that presents the HTML document the way you want, you're done. Just be sure that your choice of layout effects doesn't aggravate those suffering from particular medical conditions:

- Those with light-triggered epilepsy can seizure when subject to blinking text or images. Sensitivity varies between the 4Hz and 59Hz frequencies, with peak sensitivity around 20Hz.

- Color perception problems are quite common – more so in males than females. Make sure your layout doesn't rely on color as the sole discriminator between different objects. The filter available at `www.visicheck.com` can show you what your page looks like to users with different color perception difficulties.

- Do not use text sizes that are too small. The minimum size should appear to be equivalent to a 10pt font, but 12pt is preferable. Note that you should not actually use `pt` or `px` units to specify font sizes, as these don't scale up when the user changes the text size in their browser. The `em` unit should be used instead.

---

[*] First published 2 Nov 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=69
[1] http://www.w3.org/TR/WAI-WEBCONTENT/
[2] http://lynx.browser.org/
[3] http://www-3.ibm.com/able/solution_offerings/hpr.html

SWT: So What?*

If you are about to undertake a major project using SWT, I suggest you think very carefully before doing so. Compared to its obvious competitor, Swing, SWT is very lacking in functionality, support and community development experience. Little wonder that there is not a lot of detailed information to be found from people who are using SWT in anger to create serious applications. There is a certain amount of fan-boy stuff[1], written by people in the first blush of initial enthusiasm, convinced that everything is "cool" and "awesome", but very little from people who have been through a significant implementation effort extending over months or years. The closest one can get to finding "veteran" users is on the `eclipse.org.swt` newsgroup. In surveying opinions on SWT from the development community, I have found that people's enthusiasm for SWT is inversely proportional to the amount of experience they have had with it.

Let me briefly outline the principle differences between SWT and Swing, at a high level:

- Sun first released Swing in 1997. It is bundled with Java and is considered the "standard" for GUI development in Java. Swing creates a GUI using only emulation - that is, Java draws the buttons, menus and other widgets on a blank window using primitive graphic operations. It entirely ignores whatever widgets are made available by the native platform, but through its pluggable "Look and Feel" facility it imitates the appearance and behavior of those widgets.

- IBM released SWT as open source in 2001, having written it to support development of the Eclipse IDE. IBM began developing Eclipse in Swing, found it unacceptably slow, and so decided to write their own widget toolkit instead. In general, SWT wraps the native widgets from the underlying platform, which is intended to give better performance than Swing, and make interfaces written with SWT indistinguishable from native applications.

Discussions of the relative merits of Swing and SWT fall tend towards religious war. SWT advocates champion SWT's fidelity to native applications, performance and efficiency. They deride Swing's responsiveness, memory consumption and complexity. Swing advocates champion Swing's maturity, power and support. They deride SWT's capabilities, quality and small developer base. Advocates from both sides consider their opponents to be of questionable parentage.

Problems In Using SWT

There are numerous obstacles for the would-be SWT programmer to overcome. Collectively, you will find them a source of great frustration.

Bugs

Unless you are developing a trivial interface, you will be forced to become very well acquainted with the Bugzilla at `eclipse.org`. As further examples, try doing a query on the Bugzilla to find the number of bugs raised by the principal developers of Azureus[2] and BitTorrent[3] - probably the two most well-known SWT applications at this time. You will see that each has raised fifty or more issues in the course of developing their products. That may be fine if you're working on an open source application without strict deadlines or resource limitations, but in a commercial context, losing so much time and effort to bugs is a major problem.

You don't want your project to have critical issues to be fixed on a time line that is beyond your control. The old open source standby of "just fix it yourself" is a non-sequitur here. In a commercial context, one is paid to advance the business interests of the client, not to overcome shortcomings in a widget toolkit. Besides, making additions to SWT requires a low-level knowledge of the behavior of five different operating systems and windowing environments, and how many people have that kind of expertise?

The fact that each bug fix must be made to work for different native implementations is a significant multiplication of effort, and the source of often lengthy delays when it comes to bug fixes and functional enhancements. This was stated by Steve Northover, the original architect of SWT, in a recent message to the `eclipse.org.swt` newsgroup. He responded to one programmer's frustrated complaints about bugs in the `Table` widget which had been outstanding for several years, in this way:

> *If you stop to think about it, we support 5 different operating systems using totally different code bases and somehow knit together and implement a portable API to all of them and we do this for free. It's a full time job, 24-7.*

This problem is an unavoidable byproduct of the architectural decision that underlies SWT – the use of native widgets necessitates the development and maintenance of numerous distinct code bases. The burden is significant and, to quote James Gosling, "a bad place to be".[4]

Limited Functionality

Those coming to SWT from a Swing background will probably be shocked by the absence of many bits of functionality that they are accustomed to having at their fingertips. For instance, Swing programmers will think nothing of having a Button widget that displays both a text label and image, and be surprised they can't do that in SWT unless the `Button` appears within a `ToolBar` or `CoolBar` *[Ed. 2006 – This issue has since been resolved]*. They will be used to attaching Borders to widgets as they see fit, using the Swing `BorderFactory`, but wonder why borders are only supported on some SWT widgets such as `Text` and `Label`. They will be accustomed to setting up input masks on text fields using the facilities on `JTextField`, but find in SWT they will have to write that themselves by listening to individual keystrokes on a `Text` widget.

Eclipse Driven Development

We do well to remember that SWT was originally developed in service of Eclipse. Now that Eclipse is open source and SWT is being touted by some as an alternative to Swing for general interface development, this heritage is turning out to be quite a burden. There is a bipartite division in issue response times that seems to be related to relevance to Eclipse. If a bug is found that effects Eclipse, then there is some chance of it being attended to in a reasonable time frame. If the bug doesn't effect Eclipse – then the situation is quite different. Such bugs appears to attract a much lower priority. And given the resource restrictions the Eclipse GUI team struggles with, getting enhancement requests done is quite an achievement. This Eclipse-centric approach to maintenance and extension is a problem when the application you're constructing is not from the same domain as Eclipse. The facilities required to construct the interface for, say, a warehousing or inventory-tracking system are different from those required to construct a programmer's IDE. The former makes demands of SWT not made by the latter – but maintenance and enhancement appears to be prioritized according to relevance to Eclipse. Therefore you'll find SWT less and less relevant the further away you stray from the programming domain.

Myths

There is a lot of urban myth and misinformation surrounding both SWT and Swing. When evaluating the relative merits of these two technologies, your first task will be to distinguish fact from opinion. There is much of the latter masquerading as the former. Below, I address a few of the common misconceptions in this area.

SWT Is Fast, Swing Is Slow

Apparently it was performance concerns with Swing that prompted IBM to begin development of SWT. It would be interesting to know if they would make this same decision now, especially given the Swing performance improvements in JDK1.5. In practice, both Swing and SWT applications can be made to appear unresponsive if you perform long-running operations in the GUI event thread (a concept shared by both) or if a big garbage collection cycle arrests the entire application. The best way to compare Swing and SWT performance would be via benchmarks, however it is difficult to construct a fair comparison that truly compares like with like when the underlying technologies differ in such fundamental ways.

SWT Exposes The Native Widgets Of The Underlying Platform

In general, SWT exposes the behavior of the native platform's own GUI widget set. However this is only part of the story. There are some inferences people tend to make based on this, that are incorrect.

Some believe that the entirety of the underlying widget's behavior is exposed through SWT. This is not necessarily so. SWT must produce the same behavior across all the platforms it caters to. If widget W has behaviors A, B and C on its native platform, but C is missing from one platform's implementation of the widget W, then only A and B are provided by W on all platforms. In other words, behavior C will be masked out on its native platform, because it was not available on all platforms. This "lowest common denominator" approach can be very limiting. For example, you would not think it a great challenge to put both an image and a text label on a button. However, unless the button is in a `Toolbar` or

`CoolBar`, you can't do it in SWT *[Ed. 2006 – This issue has since been resolved]*. This is because it's not permitted on one of the platforms that SWT supports, therefore it can't be available on any of them. Every few weeks, somebody posts a message to the SWT newsgroup wanting to know how to do this, and is surprised to find that they can't ... they have to write their own button widget if they want that functionality.

However, the situation is not that simple. Sometimes the "lowest common denominator" is augmented using emulation in SWT. In other words, somebody has determined that the lowest common denominator is simply not acceptable, and those platforms where the behavior is not available natively have that behavior added on by SWT itself. In some cases this extends to emulation of an entire widget. For example, Motif has no tree widget. Rather than hide the tree widget on all platforms, SWT emulates the entire tree widget for Motif.

There are both advantages and disadvantages to SWT's partial exposure of native widgets. On the up side, you get fidelity to platform appearance and behavior. On the down side, that fidelity may not extend to the inclusion of features outside of the LCD. Further on the down side, not only do you get the native widget's behavior, you also get its bugs. On the up side, sometimes SWT can compensate for those bugs so that they appear fixed to the SWT user.

Platform Fidelity Increases Usability

The rationalization that SWT proponents constantly offer for attaching such importance to absolute platform fidelity is that it increases usability. SWT is meant to offer greater platform fidelity than Swing, which makes the usability of SWT applications better. I believe this argument is specious, for several reasons.

First, this argument gets voiced by programmers, not users. This is significant because what is important to programmers is not necessarily important to the general user population. There is also the possibility of programmers letting their technical convictions influence their perception of usability. Consider, it was feedback from programmers that drove the development of SWT to begin with. In the forward to "SWT: The Standard Widget Toolkit", Erich Gamma states:

> *I was part of the team with the mission to build a Java based integrated development environment for embedded applications that was shipped as the IBM VisualAge/MicroEdition. ... We felt pretty good about what we had achieved! However, our early*

adopters didn't feel as good as we did... they complained about the performance and most importantly about the fact that the IDE didn't look, feel and respond like a native Windows application. Some of the performance problems were our fault and some of them could be attributed to Swing. The performance problems didn't bother us that much; they could be engineered away over time. What worried us more was the non-native criticism. While we could implement a cool application in Swing that runs on Windows, we couldn't build a true Windows application. Fixing this problem required more drastic measures.

So SWT sprung from an IDE development effort, and the feedback of the IDE's early adopters - who are themselves programmers. I suspect that the issue of platform fidelity is of very little significance to non-programmers. Personally, I have seen no evidence that whatever discrepancies exist between Swing's emulation of Windows and the native Windows appearance make any appreciable difference in usability at all. Many users don't even notice, and those that do only have a vague awareness that something is a bit different about the application, but they're not quite sure what.

Second, due to the LCD effect already described, SWT often doesn't expose the exact behavior or appearance of the native widget set. Where is the evidence that the difference in fidelity between the SWT version of widgets and the Swing emulation of those widgets actually results in a difference in usability? In fact, there is much to suggest that it is not the case. Consider the success of applications such as iTunes for Windows, QuickTime, Winamp and the Firefox browser. All of these have interfaces very different from that of native Windows applications – yet they are successfully used by even novice Windows users. When users upgrade from one version of Windows to another, say from 2000 to XP, there are numerous cosmetic differences in the interface presented, but do they suddenly find themselves lost and unable to use the applications? No, of course not. The reason is that minor aesthetics are not key determinants of usability. Overall interface structure, task orientation and affordance are the key factors. Whether a button has a 3-pixel wide or 2-pixel wide shadow is not important. As long as a user can recognize the controls presented to them, and those controls behave in a predictable way, then usability is unaffected.

Finally, if usability and platform fidelity are so inextricably linked, what are we to make of the Flat Look part of SWT – that subset which creates interfaces which are similar to web pages in appearance but exhibit greater functionality? They are entirely unlike anything in any of the native platforms that SWT supports. If you've seen the PDE in Eclipse, you've seen Flat Look. If the claim that platform fidelity is linked to usability is true, shouldn't Flat Look interfaces be usability nightmares? The inconsistency between philosophy and implementation is puzzling.

SWT Is Quicker To Learn Than Swing

SWT enthusiasts claim that it is easier to learn than Swing. Having been through the learning curve for both, I have not found this to be the case. There are two main aspects to the ease of learning for any technology – the difficulty of the technical concepts themselves, and the way those concepts are taught. Conceptually, there is a significant overlap between SWT and Swing. Component hierarchies, layout managers, threading and separation of data from presentation are concepts present in both. The basic selection of built-in widgets and layouts is much the same also. The real differentiator is the quality and quantity of instructional material available. The Javadoc for SWT is sparse, the remaining knowledge has to be pieced together from articles, code snippets and asking questions on the SWT newsgroup. There are perhaps a half dozen books on SWT available. Beyond that, you need to look at the SWT code itself and reverse engineer an understanding of what's going on. The situation with Swing is very different. The Javadoc is extensive, there is a vast amount of tutorial information available online, and a large number of books are dedicated to the topic. Therefore learning Swing is generally easier than learning SWT, because of the greater amount of plain English information available.

Limited Third Party Widget Selection Is A Good Thing

Any comparison of SWT and Swing must unearth the fact that there is next to nothing in the way of third part widgets available for SWT, but there are a number of such offerings available for Swing. This can have a profound effect on programmer productivity, forcing one to write by hand what might otherwise be available off the shelf for considerably less cost.

Probably the most desperate pro-SWT argument I've heard to date is the claim that this reduced selection of COTS widgets is a good thing because it reduced the opportunities for programmers to do the wrong thing. If there is a wide selection of widgets available, the argument goes, then programmers will fill their interfaces with every cute widget they can get their hands on. This is not a problem when using SWT, as few such widgets are available in the first place.

The argument is so ridiculous as to beggar belief, but it is one I have heard SWT zealots voice, in a desperate attempt to rationalize their ideological convictions. Its main failing is to confuse widget availability and widget usage. The usability of an interface is not a function of how many different types of widgets it contains, but of the way those widgets are organized and used in the interface. A good interface designer knows that novel widgets may confuse users unfamiliar with them, and so does not employ them unless they offer a radical functional improvement in return for lesser intuitiveness. A bad interface designer will construct an interface with poor usability regardless of how few widgets they have at their disposal. To understand why, consider the following analogy.

Suppose you take a good artist and a bad artist, give them each a palette of one thousand colors then ask them to paint a picture. The good artist produces a work of art, the bad artist an eyesore. Now, in an attempt to make it harder for the bad artist to do the wrong thing, you restrict them both to a palette of ten colors. What results? The good artist produces another work of art, perhaps less subtle than the first, and the bad artist produces another eyesore, just with less variation in hue. By restricting the color selection, you haven't made it harder for the bad artist to create a mess, you've just made it more difficult for the good artist to use their talent to the fullest. The worth of the final painting is a function of the artist's talent much more than it is the availability of colors. So it is too with user interfaces. The usability of the interface is mostly a function of the designer's talent and experieoplnce, not the number of widgets available to them.

Conclusion

There has been a revival of interest lately in rich client interfaces. It seems that the obsession with web applications that the industry has experienced in recent years may be starting to thaw. It is finally being appreciated that it is not OK to squeeze all interaction through the restrictions currently imposed by web browsers. Even though programmers may be temporarily enamored of web-based development, their enthusiasm is not necessarily shared by the user population who must struggle with the results of their IT department's technical and ideological enthusiasms.

So now it is time for programmers to impose their technical preferences regarding rich client interfaces upon an unsuspecting user group, for which they will need some ostensible justification - hence the cattle call to SWT, and the unsubstantiated claims in its favor.

For those interested in what actually benefits their organization, rather than what looks best on their CV and is "cool", there is really no competition between Swing and SWT. SWT is simply not ready for generalized interface development, and given that its development lags behind Swing some seven years, one has to wonder how its use and continued development can be rationalized.

If you are developing a rich interface in Java, and considering both SWT and Swing, I urge you to consider the following issues:

- If you believe that the greater platform fidelity of SWT will make for a more usable application, what actual evidence do you have to support that conclusion? Have you put both in front of your user population?
- It's hard to find good GUI developers. Finding good GUI developers with SWT skills is even harder. Where are you going to find the staff to develop your GUI in SWT? If you anticipate getting Swing developers to cross-train in SWT, get ready for staff turnover. Taking a Swing developer and giving them SWT is like taking someone used to riding a Harley Davidson and giving them a Vespa motor scooter. They're not likely to be delighted.
- How close is your target GUI to the Eclipse GUI? Be aware that every time you step even a little way beyond the functional demands of Eclipse, you are on your own. You will likely have to start writing custom widgets in order to get the behavior you want. Can your organization justify spending time and money writing widgets that in Swing, would be available off the shelf?
- Due to the bugs and shortcomings in SWT, your developers will be working with a lowered productivity, and so you should expect project delays and/or increased resource requirements. Can your organization justify this extra investment?
- Before deciding that Swing applications are slow and ugly, take the time to look at products like Netbeans and GUI libraries such as JIDE. I have heard people voice these opinions, having not looked at Swing since the days of AWT.

- Is your source of information about SWT the blogs of novice GUI developers, or those who have had only a fleeting encounter with SWT. Let me suggest you subscribe to the SWT newsgroup and mailing list where you will get the perspective of those who have been struggling with it for a longer period of time, and are past that initial phase of enthusiasm.

Of course, just because SWT is the technically inferior solution doesn't mean that it will go away. Hype, marketing, vendor over-enthusiasm and managerial stupidity can propel a second-rate solution to prominence. This may yet prove to be the case for SWT.

SWT Resources

- *Professional Java Native Applications with SWT/JFace*, J.L. Guojie
- *Definitive Guide to SWT and JFace*, R. Harris, R. Warner
- *SWT/JFace in Action*, M. Scarpino et.al.
- *SWT Developers Notebook*, T. Hatton
- *Developing Quality Plugins for Eclipse*, E. Clayberg
- *Contributing to Eclipse*, E. Gamma, K. Beck, Addison Wesley, 2004
- *SWT: The Standard Widget Toolkit*, Volume 1, S. Northover, M. Wilson, Addison Wesley, 2004
- *SWT Designer*, http://www.swt-designer.com/
- *SWT Sightings*, http://www.oneclipse.com/Members/admin/news/swt-sightings

* First published 24 Apr 2005 at http://www.hacknot.info/hacknot/action/showEntry?eid=74
[1] http://blogs.bytecode.com.au/glen/2005/02/12/1108169609271.html
[2] http://azureus.sourceforge.net/
[3] http://www.bittorrent.com/
[4] http://www.builderau.com.au/program/work/0,39024650,39176462,00.htm

Debugging and Maintenance

Debugging 101*

"An interactive debugger is an outstanding example of what is not needed – it encourages trial-and-error hacking rather than systematic design, and also hides marginal people barely qualified for precision programming."– Harlan Mills

Recently, a colleague and I were working together to resolve a bug in a piece of code she had just written. The bug resulted in an exception being thrown and looking at the stack trace, we were both puzzled about what the root cause might be. Worse yet, the exception originated from within an open source library we were using. As is typical of open source products, the documentation was sparse, and wasn't providing us with very much help in diagnosing the problem before us. It was beginning to look like we might have to download the source code for this library and start going through it – a prospect that appealed to neither of us.

As a last resort before downloading this source code, I suggested that we try doing a web search on the text of the exception itself, by copying the last few lines of the stack trace into the search field for a web search engine. I hoped the search results might include pages from online forums where someone else had posted a message like "I'm seeing the following exception, can anyone tell me what it means?", followed by all or part of the stack trace itself. If the original poster had received a helpful response to their query, then perhaps that response would be helpful to us too.

My colleague, who is reasonably new to software development, was surprised by the idea and commented that it was something she would never have thought to try. Her response got me to thinking about debugging techniques in general, and the way we acquire our knowledge of them.

Reflecting on my formal education in computer science, I cannot recall a single tutorial or lecture that discussed how I should go about debugging the code that I wrote. Mind you, I cannot remember much of anything about those lectures, so perhaps it really was addressed and I've simply forgotten. Even so, it seems that the topic of debugging is much neglected in both academic and trade discussions. Why is this?

It seems particularly strange when you consider what portion of their time the average programmer spends debugging their own code. I've not measured it for myself, but I wouldn't be surprised if one third or more of my day was spent trying to figure out why my code doesn't behave the way I expected. It seems strange that I never learnt in any structured way how to debug a program. Everything I know about debugging has been acquired through experience, trial and error, and from watching others. Unless my experience is unique, it seems that debugging techniques should be a topic of vital interest to every developer. Yet some developers seem almost embarrassed to discuss it.

I suspect the main reason for this is hubris. The ostensible ability to write bug-free code is a point of pride for many programmers. Displaying a knowledge of debugging techniques is tantamount to admitting imperfection, acknowledging weakness, and that really sticks in the craw of those developers who like to think of themselves as "l337 h4x0r5". But by avoiding the topic, we lose a major opportunity to learn methods for combating our inevitable human weaknesses, and thereby improving the quality of the work we do.

So I've taken it upon myself to list the main debugging techniques that I am aware of. For many programmers, these techniques will be old hat and quite basic. But even for veteran debuggers there may be value in bringing back to mind some of these tried and true techniques. For others, there might be one or two methods that you hadn't thought of before. I hope they save you a few hours of frustrating fault-finding.

General Principles

Regardless of the specific debugging techniques you use, there are a few general principles and guidelines to keep in mind as your debugging effort proceeds.

Reproduce

The first task in any debugging effort is to learn how to consistently reproduce the bug. If it takes more than a few steps to manually trigger the buggy behavior, consider writing a small driver program to trigger it programmatically. Your debugging effort will proceed much more quickly as a result.

Progressively Narrow Scope

There are two basic ways to find the origin of a bug – *brute force* and *analysis*. *Analysis* is the thoughtful consideration of a bug's likely point of origin, based on detailed knowledge of the code base. A *brute force* approach is a largely mechanical search along the execution path until the fault is eventually found.

In practice, you will probably use a combination of both methods. A preliminary analysis will tell you the area of the code most likely to contain the bug, then a brute force search within that area will locate it precisely.

Purists may consider any application of the brute force approach to be tantamount to hacking. It may be so, but it is also the most expedient method in many circumstances. The quickest way to search the path of execution by brute force is to use a binary search, which progressively divides the search space in half at each iteration.

Avoid Debuggers

In general, I recommend you avoid symbolic debuggers of the type that have become standard in many IDEs. Debuggers tend to produce a very fragile debugging process. How often does it happen that you spend an extended period of time carefully stepping through a piece of code, statement by statement, only to find at the critical moment that you accidentally "step over" rather than "step into" some method call, and miss the point where a significant change in program state occurs? In contrast, when you progressively add trace statements to the code, you are building up a picture of the code in execution that cannot be suddenly lost or corrupted. This repeatability is highly valuable – you're monotonically progressing towards your goal.

I've noticed that habitual use of symbolic debuggers also tends to discourage serious reflection on the problem. It becomes a knee-jerk response to fire up the debugger the instant a bug is encountered and start stepping through code, waiting for the debugger to reveal where the fault is.

That said, there are a small number of situations where a debugger may be the best, or perhaps only, method available to you. If the fault is occurring inside compiled code that you don't have the source code for, then stepping through the just-in-time decompiled version of the executable may be the only way of subjecting the faulty code to scrutiny. Another instance where a debugger can be useful is in the case of memory overwrites and corruption, as can occur when using languages that permit direct memory manipulation, such as C and C++. The ability most debuggers provide to "watch" particular memory segments for changes can be helpful in highlighting unintentional memory modifications.

Change Only One Thing At A Time

Debugging is an iterative process whereby you make a change to the code, test to see if you've fixed the bug, make another change, test again, and so on until the bug is fixed. Each time you change the code, it's important to change only one aspect of it at a time That way, when the bug is eventually fixed, you will know exactly what caused it – namely, the very last thing you touched. If you try changing several things at once, you risk including unnecessary changes in your bug fix (which may themselves cause bugs in future), and diluting your understanding of the bug's origin.

Technical Methods

Debugging is a manually intensive activity more like solving logic problems or brain teasers than programming. You will find little use for elaborate tools, instead relying on a handful of simple techniques intelligently applied.

Insert Trace Statements

This is the principle debugging method I use. A trace statement is a human readable console or log message that is inserted into a piece of code suspected of containing a bug, then generally removed once the bug has been found. Trace statements not only trace the path of execution through code, but the changing state of program variables as execution progresses. If you have used Design By Contract (see “Introduce Design By Contract” below) diligently, you will already know what portion of the code to instrument with trace statements. Often it takes only half a dozen or so well chosen trace statements to pinpoint the cause of your bug. Once you have found the bug, you may find it helpful to leave a few of the trace statements in the code, perhaps converting console messages into file-based logging messages, to assist in future debugging efforts in that part of the code.

Consult The Log Files Of Third Party Products

If you're using a third party application server, servlet engine, database engine or other active component then you'll find a whole heap of useful information about recently experienced errors in that component's own log files. You may have to configure the component to log the sort of information you're interested in. In general, if your bug seems to involve the internals of some third party product that you don't have the source code for (and so can't instrument with trace statements), see if the vendor has supplied some way to provide you with a window into the product's internal operation. For example, an ORM library might produce no console output at all by default, but provide a command line switch or configuration file property that makes it output all SQL statements that it issues to the database.

Search The Web For The Stack Trace

Cut the text from the end of a stack trace and use it as a search string in the web search engine of your choice. Hopefully this will pick up questions posted to discussion forums, where the poster has included the stack trace that they are seeing. If someone posted a useful response, then it might relate to your bug. You might also search on the text of an error message, or on an error number. Given that search engines might not discover dynamically generated web pages in discussion forums, you might also find it profitable to identify web sites likely to host discussions pertaining to your bug, and use the site's own search facilities in the manner just described.

Introduce Design By Contract

In my opinion, DBC is one of the best tools available to assist you in writing quality code. I have found rigorous use of it to be invaluable in tracking down bugs. If you're not familiar with DBC, think of it as littering your code with assertions about what the state of the program should be at that point, if everything is going as you expect it to. These assertions are checked programmatically, and an exception thrown when they fail. DBC tends to make the point of program failure very close to the point of logical error in your code. This avoids those frustrating searches where a program fails in function C, but the actual error was further up the call chain in function A, which passed on faulty values to function B, which in turn passed the values to function C, which ultimately failed. It's best to use

DBC as a means of bug prevention, but you can also use it as a means of preventing bug recurrence. Whenever you find a bug, litter the surrounding code with assertions, so that if that code should ever go wrong again, a nearby assertion will fail.

Wipe The Slate Clean

Sometimes, after you've been hunting a bug for long enough, you begin to despair of ever finding it. There may be an overwhelming number of possible sources yet to explore, or the behavior you're observing is just plain bizarre. On such occasions it can be useful to wipe the slate clean and start again. Create an entirely new mini-application whose sole function is to demonstrate the presence of your bug. If you can write such a demo program, then you're well on your way to tracking down the cause of the bug. Now that you have the bug isolated in your demo program, start removing potentially faulty components one by one. For example, if your demo program uses some database connection pooling library, cut it out and run the program again. If the bug persists, then you've just identified one component that doesn't contribute to the buggy behavior. Proceed in that manner, stripping out as many possible fault sources as you can, one at a time. When you remove a component that makes the bug disappear, then you know that the problem is related to the last component you removed.

Intermittent Bugs

A bug that occurs intermittently and can't be consistently reproduced is the programmer's bane. They are often the result of asynchronous competition for shared resources, as might occur when multiple threads vie for shared memory or race for access to a local variable. They can also result from other applications competing for memory and I/O resources on the one machine.

First, try modifying your code so as to serialize any operations occurring in parallel. For example, don't spawn N threads to handle N calculations, but perform all N calculations in sequence. If your bug disappears, then you've got a synchronization problem between the blocks of code performing the calculations. For help in correctly synchronizing your threads, look first to any support for threading that is included in your programming language. Failing that, look for a third party library that supports development of multi-threaded code.

If your programming language doesn't provide guaranteed initialization of variables, then uninitialized variables can also be a source of intermittent bugs. 99% of the time, the variable gets initialized to zero or `null` and behaves as you expected, but the other 1% of the time it is initialized to some random value and fails. A class of tools called "System Perturbers"[1] can assist you in tracking down such problems. Such tools typically include facility for zero-filling memory locations, or filling memory with random data as a way of teasing out initialization bugs.

Exploit Locality

Research shows that bugs tend to cluster together. So when you encounter a new bug, think of those parts of the code in which you have previously found bugs, and whether nearby code could be involved with the present bug.

Read The Documentation

If all else fails, read the instructions. It's remarkable how often this simple step is foregone. In their rush to start programming with some class library or utility some developers will adopt a trial-and-error approach to using a new API. If there is little or no API documentation then this may be an appropriate approach. But if the API has some decent programmer-level documentation with it, then take the time to read it. It's possible that your bug results from misuse of the API and the underlying code is failing to check that you have obeyed all the necessary preconditions for its use.

Introduce Dummy Implementations And Subclasses

Software designers are sometimes advised to "write to interfaces". In other words, rather than calling a method on a class directly, call a method on an interface that the class implements. This means that you are free to substitute in a different class that implements the same interface, without needing to change the calling code. While dogmatic application of this guideline can result in a proliferation of interfaces that are only implemented once, it does point to a useful debugging technique. If the outcome of the collaboration between several objects is buggy, look to the interfaces that the participating objects implement. Where an object is invoked only via interfaces, consider replacing the object with a simple, custom object of your own that is hard-wired to perform correctly under very specific circumstances. As long as you limit your testing to the circumstances that you know your custom object handles correctly, you know that any buggy behavior you subsequently observe must be the fault of one of the other objects involved. That is, you've eliminated one potential source of the bug. You can achieve a similar effect by substituting a custom subclass of a participant class, rather than a custom implementation of an interface.

Recompile And Relink

A particularly nasty type of bug arises from having an executable image that is a composite of several different compile and/or relink operations. The failure behavior can be quite bizarre and it can appear that internal program state is being corrupted "between statements". It's like gremlins have crept into your code and started screwing around with memory.

Most recently, I have encountered this bug in Java code when I change the value of string constants. It seems the compiler optimizes references to string constants by inserting them literally at the point of reference. So the constant value is copied to multiple class files. If you don't regenerate all those class files after changing the string constant, those class files not regenerated will still contain the old value of that constant. Performing a complete recompilation prevents this from occurring. Finally, set the compiler to include debugging information in the generated code, and set the compiler warning level to the maximum.

Probe Boundary Conditions And Special Cases

Experienced programmers know that it's the limits of an algorithmic space that tend to get forgotten or mishandled, thereby leading to bugs. For example, the procedure for deleting records 1 to N might be slightly different from the procedure for deleting record 0. The algorithm for determining if a given year is a leap year is slightly different if the year is divisible by 400. Breaking a string into a list of space-separated words requires consideration of the cases where the string contains only one word, or is empty. The tendency to code only the general case and forget the special cases is a very common source of error.

Check Version Dependencies

One of the most obscure sources of a bugs is the use of incompatible versions of third party libraries. It is also one of the last things to check when you've exhausted other debugging strategies. If version 1.0.2 of some library has a dependency on version 2.4 of another library, but you supply version 2.5 instead, the results may be subtle failures that are difficult or impossible to diagnose. Look particularly to any libraries that you have upgraded just prior to the appearance of the bug.

Check Code That Has Changed Recently

When a bug suddenly appears in functionality that has been working for some time, you should immediately wonder what has recently changed in the code base that might have caused this regression. This is where your version control system comes into its own, providing you with a way of looking at the change history of the code, or recreating successively older versions of the code base until you get one in which the regression disappears.

Don't Trust The Error Message

Normally you scrutinize the error messages you get very carefully, hoping for a clue as to where to start your debugging efforts. But if you're not having any luck with that approach, remember that error messages can sometimes be misleading. Sometimes programmers don't put as much thought into the handling and reporting of error conditions as one would like, so it may be wise to avoid interpreting the error message too literally, and to consider possibilities other than the ones it specifically identifies.

Graphics Bugs

There are a few techniques that are particularly relevant when working on GUIs or other graphics-related bugs. Check if the graphics pipeline you are using includes a debugging mode – a mode which slows down graphics operations to a speed where you can observe individual drawing operations occurring. This mode can be very useful for determining why a sequence of graphic operations don't combine to give the effect you expected.

When debugging problems with layout managers, I like to set the background colors of panels and components to solid, contrasting colors. This enables you to see exactly where the edges of the components are, which highlights the layout decisions made by the layout managers involved.

Psychological Methods

I think it's fair to say that the vast majority of bugs we encounter are a result of our own cognitive limitations. We might fail to fully comprehend the effects of a particular API call, forget to free memory we've reserved, or simply fail to translate our intent correctly into code. Indeed, one might consider debugging to be the process of finding the difference between what you instructed the machine to do, and what you *thought* you instructed the machine to do. So given their basis in faulty thinking, it makes sense to consider what mental techniques we can employ to think more effectively when hunting bugs.

Wooden Indian

When you're really stuck on a bug, it can be helpful to grab a colleague and explain the bug to them, together with the efforts you've made so far to hunt down its source[2]. It may be that your colleague can offer some helpful advice, but this is not what the technique is really about. The role of your colleague is mainly just to listen to your description in a passive way. It sometimes happens that in the course of explaining the problem to another, you gain an insight into the bug that you didn't have before. This may be because explaining the bug's origin from scratch forces you to go back over mental territory that you haven't critically examined, and challenge fundamental assumptions that you have made. Also, by verbalizing you are engaging different sensory modalities which seems to make the problem "fresh" and revitalizes your examination of it.

Don't Speculate

Arthur C. Clarke once wrote "Any sufficiently advanced technology is indistinguishable from magic." And so it is for any sufficiently mysterious bug. One of the greatest traps you can fall into when debugging is to resort to superstitious speculation about its cause, rather than engaging in reasoned enquiry[3]. Such speculation yields a trial-and-error debugging effort that might eventually be successful, but is likely to be highly inefficient and time consuming. If you find yourself making random tweaks without having some overall strategy or approach in mind, stop straight away and search for a more rational method.

Don't Be Too Quick To Blame The Tools

Perhaps you've had the embarrassing experience of announcing "it must be a compiler bug" before finding the bug in your own code. Once you've done it, you don't rush to judgment so quickly in the future. Part of rational debugging is realistically assessing the probability that there is a bug in one of the development tools you are using. If you are using a flaky development tool that is only up to its beta release, you would be quite justified in suspecting it of error. But if you're using a compiler that has been out for several years and has proven itself reliable in the field over all that time, then you should be very careful you've excluded every other possibility before concluding that the compiler is producing a faulty executable.

Understand Both The Problem And The Solution

It's not uncommon to hear programmers declare "That bug disappeared" or "It must've been fixed as a side-effect of some other work". Such statements indicate that the programmer isn't seeking a thorough understanding of the cause of a bug, or its solution, before dismissing it as no longer in need of consideration. Bugs don't just magically disappear. If a bug seems to be suddenly fixed without someone having deliberately attended to it, then there's a good chance that the fault is still somewhere in the code, but subsequent changes have changed the way it manifests. Never accept that a bug has disappeared or fixed itself. Similarly, if you find that some changes you've made appear to have fixed a bug, but you're not quite sure how, don't kid yourself that the fix is a genuine one. Again, you may simply have changed the character of the bug, rather than truly fixing its cause.

Take A Break

In both bug hunting and general problem solving I've experienced the following series of events more times than I can remember. After struggling with a problem for several hours and growing increasingly frustrated with it, I reach a point where I'm too tired to continue wrestling with it, so I go home. Several choice expletives are muttered. Doors are slammed. The next morning, I sit down to continue tackling the problem and the solution just falls out in the first half hour.

Many have noted that solutions come much easier after a period of intense concentration on the problem, followed by a period of rest.

Whatever the underlying mechanism might be, if you have similar experiences its worth remembering them when you're faced with a decision between bashing your head against a problem for another hour, or having a rest from it.

Another way to get a fresh look at a piece of code you've been staring at for too long is to print it out and review it off the paper. We read faster off paper than off the screen, so this may be why it's slightly easier to spot an error in printed code than displayed code.

Consider Multiple Causes

There is a strong human tendency to oversimplify the diagnoses of problems, attributing what may be multi-causal problems to a single cause. The simplicity of such a diagnosis is appealing, and certainly easier to address. The habit is encouraged by the fact that many bugs really are the result of a single error, but that is by no means universally the case.

Bug Prevention Methods

"Prevention is better than cure," goes the maxim; as true of sicknesses in code as of sicknesses in the body. Given the inevitability and cost of debugging during your development effort, it's wise to prepare for it in advance and minimize it's eventual impact.

Monitor Your Own Fault Injection Habits

After time you may notice that you are prone to writing particular kinds of bugs. If you can identify a consistent weakness like this, then you can take preventative steps. If you have a code review checklist, augment the checklist to include a check specifically for the type of bug you favor. Simply maintaining an awareness of your "favorite" defects can help reduce your tendency to inject them.

Introduce Debugging Aids Early

Unless you've somehow attained perfection prior to starting work on your current project, you can be confident that you have numerous debugging efforts in store before you finish. You may as well make some preparation for them now. This means inserting logging statements as you proceed, so that you can selectively enable them later, before augmenting them with bug-specific trace statements. Also think about the prime places in your design to put interfaces. Often these will be at the perimeters of significant subsystems. For example, when implementing client-server applications, I like to hide all client contact with the server behind interfaces, so that a dummy implementation of the server can be used in place of the server, throughout client development. It's not only a convenient point of interception for debugging efforts, but a development expedient, as the test-debug cycle can be significantly faster without the time cost of real server deployment and communication.

Loose Coupling And Information Hiding

Application of these two principles is well known to increase the extensibility and maintainability of code, as well as easing its comprehension. Bear in mind that they also help to prevent bugs. An error in well modularized code is less likely to produce unintended side-effects in other modules, which obfuscates the origin of the bug and impedes the debugging effort.

Write A Regression Test To Prevent Reoccurrence

Once you've fixed the bug it's a good idea to write a regression test that exercises the previously buggy code and checks for correct operation[4]. If you wish, you can write that regression test before having fixed the bug, so that a successful bug fix is indicated by the successful run of the test.

Conclusion

If you spend enough time debugging, you start to become a bit blasé about it. It's easy to slip into a rut and just keep following the same patterns of behavior you always have, which means that you never get any better or smarter at debugging. That's a definite disadvantage, given how much of the average programmer's working life is consumed by it. There's also a tendency not to examine debugging techniques closely or seriously, as debugging is something of a taboo topic in the programming community.

It's wise to acknowledge your own limitations up front, the resultant inevitability of debugging, and to make allowances for it right from the beginning of application development. It's also worth beginning each debugging effort with a few moments of deliberate reflection, to try and deduce the smartest and quickest way to find the bug.

[*] First published 17 Apr 2006 at http://www.hacknot.info/hacknot/action/showEntry?eid=85
[1] *Rapid Development*, Steve McConnell, Microsoft Press, 1996
[2] *The Pragmatic Programmer*, A. Hunt and D. Thomas, Addison-Wesley, 2000
[3] *Code Complete*, Steve McConnell, Microsoft Press, 1993
[4] *Writing Solid Code*, Steve Maguire, Microsoft Press, 1993

Spare a Thought for the Next Guy*

I just had a new ISDN phone line installed at my house. It was an unexpectedly entertaining event, and provided the opportunity for some reflection on the similarity between the problems faced by software developers and those in other occupations.

The installation was performed by a technician who introduced himself in a Yugoslavian brogue as "Ranko." Ranko looked at the existing phone outlets, declared that it would be a straight forward job and should take 30 - 45 minutes.

I sat down to read a book and let Ranko go about his work.

Things seemed to be going well for him until 15 minutes into the procedure when I heard some veiled mutterings coming from the kitchen. Putting my book down to listen more carefully, I heard Ranko talking to himself in angry tones - "What have they done? What have they done to poor Ranko?" (he had an unusual habit of referring to himself in the third person).

Curious, I sauntered into the kitchen on the pretence of making myself a cup of coffee.

I found Ranko still muttering away and staring in angry disbelief at the display of some instrument. Before him were a half dozen cables spewing out of the wall like so many distended plastic intestines set loose from the house's abdominal cavity. Ranko asked if he could get up into the ceiling cavity of the house, and I assented – pointing him in the direction of the access cover. He strode outside to his van and reappeared in my front door a few moments later with a step ladder under one arm.

I returned to my reading while he pounded around above me. Shortly I heard a few exclamations of "Bloody hell!" followed by more thumping. After a brief pause, there came a series of "You bloody idiots!" / "Bastards!" two-shots in rapid succession, punctuated by some unnecessarily loud pounding of feet upon the ceiling joists. Underneath, I listened with growing amusement, choking back laughter with one hand over my mouth.

For the next 10 minutes or so I was lost in my reading, and looked up in surprise to find Ranko standing in front of me looking slightly disheveled but rather proud of himself.

"I have found the problem" he declared proudly, and proceeded to explain. It appeared the previous residents of the house had self-installed one of the telephone extensions in my house. Rather than daisy-chain the additional outlet on from another outlet, they had simply spliced into the phone line up in the ceiling cavity and run cabling from the splice point to the new outlet. This was easier for them than daisy-chaining, as it halved the number of times they had to run a phone cable through a wall cavity.

But for future technicians, it meant that any wiring changes of the type Ranko was attempting would necessitate access to the ceiling cavity where the splice-point was located. If done in daisy-chain style, as is regular practice amongst phone technicians, the wiring changes could've been done without having to ascend into the crawl space above. The job took nearly twice as long as what Ranko initially estimated, because he had the unexpected tasks of diagnosing the problem with the existing installation, determining the location of the splice and then working around it.

Sound familiar?

Ranko has experienced the same problem that maintenance programmers face every day. We estimate the duration of a maintenance task based on some assumptions about the nature of the artifact we will be altering. We begin the maintenance task, only to find that those assumptions don't hold, due to some unexpected shortcuts taken by those that came before us. Then we have to develop an understanding of those shortcuts, before we can perform our maintenance task.

And if we chose to work around the shortcut rather than fix it, future maintenance programmers will have the same problems. And so a short-term expediency made by a programmer long ago becomes the burden of every programmer that follows.

And the very need to make assumptions at all stems from the absence of any information about the morphology of the existing system. Those that hack into the phone line are not of the nature to document their efforts, nor to keep existing documentation up to date.

So next time you're under dead line pressure and have your fingers poised above the keyboard ready to take a shortcut – spare a thought for the next guy who will have to deal with that shortcut. He might be you.

[*] First published 26 Apr 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=52

Six Legacy Code AntiPatterns*

I recently began work on a J2EE project – a workflow assistance tool that has been under development for a few years. The application is totally new to me and yet is immediately familiar, for it bears the scars and wounds so common to a legacy system. Browsing through the code base and playing with the GUI, the half dozen legacy code anti-patterns that leave me with déjà vu are listed below. How many do you recognize?

Nadadoc

The Javadoc has been written in a perfunctory, content-free manner, giving rise to what I call *Nadadoc*. Here's an example of Nadadoc:

```
/**
  * Process an order
  *
  * @param orderID
  * @param purchaseID
  * @param purchaseDate
  * @return
  */
  public int processOrder(
    int orderID, int purchaseID, Date purchaseDate);
```

Just enough text is used to assuage any niggling professionalism the author might be experiencing, without the undertaking the burden of having to communicate useful information to the reader. Commenting of code is an afterthought, achieved by invoking the IDE facility for generating a Javadoc template and performing some token customization of the result.

Abandoned Framework

With school boy enthusiasm, the original authors have decided they know enough about their application domain to build a framework for the construction of similar applications, the first use of which will be the product they are trying to write. Such naivety is driven by grand notions of reuse not yet tarnished by contact with the real world. Classes constructed early in this project are so insanely generic that even fundamental types such as `java.util.Enumeration` are rewritten with bespoke versions that are ostensibly more general purpose. Classes constructed later in the project, after the team realizes that constructing a framework within the time allowed is totally infeasible, are application specific hack-fests.

GUI - Designed By Programmers And Written By Borland

Software developers seem to ubiquitously suffer the self-deception that it is easy to design a good user interface. Perhaps they confuse the ability to program a GUI with the ability to design one. Perhaps the commonality of GUIs leads them to think "everyone's doing it, so it must be easy." In any case, you can often spot a GUI designed by programmers at a glance. This is certainly the case with my current project. Common usability guidelines are violated everywhere - no keyboard access to fields, no keyboard accelerators, group boxes around single controls, no progress indicators for long operations, illogical and misaligned layouts.

At the code level, the story is even worse. Many elements of the UI have been generated by the GUI builder in an IDE – in this case JBuilder. Although it is possible to generate semi-acceptable code from these things, they are rarely used to good effect. When the default control names and layout mechanisms are used, the generated code becomes a real maintenance burden, consisting of a complex combination of components with names like `panel7`, `label23` and the like.

Oral Documentation Is Mostly Laughter

If you can't be bothered writing documentation, the lads at *Fantasy Central* (otherwise known as XP-land) have provided you with a ready-made *out* in the form of the oxymoron "oral documentation". When maintenance programmers ask "Where's the documentation?" you need only say (preferably with smug self assurance) "We use oral documentation."

The developers of this system relied very heavily on oral documentation, and there are just a few problems with it that the XP dreamers generally neglect to mention:

- The documentation set becomes self-referential. If you ask John about component X, he'll refer you to Darren, who refers you to James, who refers you back to John. Not because they don't have the answers, but because explaining the inner workings of systems they've left behind is boring.

- Parts of the documentation set keep walking out the door due to attrition. Some chapters are unavailable due to illness.
- The documentation fades rather quickly. As developers move on and become ensconced in new projects, the details of the projects they've left behind quickly fade.
- Certain pages in an oral documentation set are bookmarked with laughter. In this system, a great many of them are so marked. The laughter disguises the embarrassment of the original developers when you uncover the hacks and shortcuts in their work. Not surprisingly, developers are loathe to discuss the details of work they know is sub-standard, and enquiries in these areas result in information that is a guilty mix of admission and excuse.

Cargo-Cult Development Idioms

When developers can't understand how the code works, they tend to add functionality by just cutting and pasting segments of existing code that appear to be relevant to their development goal. There develops a series of application-specific idioms that are justified with the phrase "that's just how we do it." No one really knows why - sufficiently detailed knowledge of the code base to choose amongst implementation alternatives on a rational basis is lost or not readily available, so the best chance of success seems to be to follow those implementation idioms already present in the code.

Architecture Where Art Thou?

Many developers are not very enthusiastic about forethought. It just delays the start of coding, and that's where the real fun is. Alas, when there is no pre-planned structure for that code it tends to grow in a haphazard, organic and often chaotic way. Rather like growing a vine - if you train the vine up a trellis, then the resulting plant exhibits at least a modicum of structure. Without the trellis, the vine wanders randomly without purpose or regularity. My current project was grown without a trellis and is riddled with weeds and straggling limbs. The original developers have, presumably against their will, attempted to document the project as if there were some intentional underlying structure. But there is too little accord and too many inconsistencies between the structure described and the reality of the code base for the one to have guided the construction of the other.

[*] First published 2 Feb 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=47

Skepticism

The Skeptical Software Development Manifesto[*]

"Argumentation cannot suffice for the discovery of new work, since the subtlety of Nature is greater many times than the subtlety of argument."
– Francis Bacon

The over-enthusiastic and often uncritical adoption of XP and Agile tenets by many in the software development community is worrying.

It is worrying because it attests to the willingness of many developers to accept claims made on the basis of argument and rhetoric alone. It is worrying because an over-eagerness to accept technical and methodological claims opens the door to hype, advertising and wishful thinking becoming the guiding forces in our occupation. It is worrying because it highlights the professional gulf existing between software engineering and other branches of engineering and science, where claims to discovery or invention must be accompanied by empirical and independently verifiable experiment in order to gain acceptance.

Without skepticism and genuine challenge, we may forfeit the ability to increase our domain's body of knowledge in a rational and verifiable way; instead becoming a group of fashion followers, darting from one popular trend to another.

What is needed is a renewed sense of skepticism towards the claims our colleagues make to improved practice or technology. To that end, and to lend a little balance to the war of assertion initiated by the Agile Manifesto[1], I would like to posit the following alternative.

The Skeptical Software Development Manifesto

We are always interested in claims to the invention of better ways of developing software. However we consider that claimants carry the burden of proving the validity of their claims. We value:

- Predictability over novelty
- Empirical evidence over anecdotal evidence
- Facts and data over rhetoric and philosophy

That is, while there is value in the items on the right, we value the items on the left more.

Our skepticism is piqued by claims and rhetoric exhibiting any of the following characteristics:

- An imprecision that does not permit further scrutiny or enquiry
- The mischaracterization of doubt as fear or cynicism
- Logical and rhetorical fallacies such as those listed below:[2]

Argumentum Ad Hominem

Reference to the parties to an argument rather than the arguments themselves.

Appeal To Ignorance

The claim that whatever has not been proved false must be true, and vice versa.

Special Pleading

A claim to privileged knowledge such as "you don't understand", "I just know it to be true" and "if you tried it, you'd know it was true." [3]

Observational Selection

Drawing attention to those observations which support an argument and ignoring those that counter it.

Begging The Question

Supporting an argument with reasons whose validity requires the argument to be true.

Doubtful Evidence

The use of false, unreasonable or unverifiable evidence.

False Generalization

The unwarranted generalization from an individual case to a general case; often resulting from their being no attempt to isolate causative factors in the individual case.

Straw-Man Argument

The deliberate distortion of an argument to facilitate its rebuttal.

Argument From Popularity

Reasoning that the popularity of a view is indicative of its truth. e.g. "everybody's doing it, so there must be something to it."

Post Hoc Argument

Reasoning of the form "B happened after A, so A caused B". i.e. confusing correlation and causation.

False Dilemma

Imposing an unnecessary restriction on the number of choices available. e.g. "either you're with us or you're against us."

Arguments From Authority

Arguments of the form "Socrates said it is true, and Socrates is a great man, therefore it must be true".

We are especially cautious when evaluating claims made by parties who sell goods or services associated with the technology or method that is the subject of the claim.

Principles Behind The Skeptical Software Development Manifesto

We follow these principles:

- Propositions that are not testable and not falsifiable are not worth much.
- Our highest priority is to satisfy the customer by adopting those working practices which give us the highest chance of successful software delivery.
- We recognize that changing requirements incur a cost in their accommodation, and that claims to the contrary are unproven. We are obliged to apprise both ourselves and the customer of the realistic size of that cost.
- It is our responsibility to identify the degree/frequency of customer involvement required to achieve success, and to inform our customer of this. Our customer has things to do other than help us write their software, so we will make as efficient use of their time as we are able.

- We recognize that controlled experimentation in the software development domain is difficult, as is achieving isolation of variables, but that is no excuse for not pursuing the most rigorous examination of claims that we can, or for excusing claimants from the burden of supporting their claims.
- Quantification is good. What is vague and qualitative is open to many interpretations.

[*] First published 19 Oct 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=30
[1] http://agilemanifesto.org/
[2] *The Demon-Haunted World*, C. Sagan and A. Druyan, Ballantine Books, 1996
[3] *How To Win An Argument*, 2nd Edition, M. Gilbert, Wiley, 1996

Basic Critical Thinking for Software Developers*

Vague Propositions

A term is called “vague” if it has a clear meaning but not a clearly demarcated scope. Many arguments on Usenet groups and forums stem from the combatants having different interpretations of a vaguely stated proposition. To avoid this sort of misunderstanding, before exploring the truth of a given proposition either rhetorically or empirically, you should first state that proposition as precisely as possible.

Consider this proposition:

P(1): Pair Programming works

If I were to voice that proposition on the Yahoo XP group[1], I would expect it to receive enthusiastic endorsement. I would also expect no one to point out that this proposition is non-falsifiable.

It is non-falsifiable because the terms "pair programming" and "works" are so vague. There are an infinite number of scenarios that I could legitimately label "pair programming", and an infinite number of definitions of what it means for that practice to "work." Any specific argument or evidence you might advance to disprove P(1) will imply a particular set of definitions for these terms, which I can counter by referencing a different set of definitions – thereby preserving P(1).

A vast number of arguments about software development techniques are no more than heated and pointless exchanges fueled by imprecisely stated propositions. There is little to be gained by discussing or investigating a non-falsifiable proposition such as P(1). We need to formulate the proposition more precisely before it becomes worthy of serious consideration.

Let’s begin by rewording P(1) to clarify what we mean by "works":

P(2): Pair Programming results in better code

Now at least we know we’re talking about code as being the primary determinant of whether pair programming works. However P(2) is now *implicitly relative*, which is another common source of vagueness. An implicitly relative statement makes a comparison with something without specifying what that something is. Specifically, it proposes that pair programming produces better code, but better code than what?

Let’s try again:

P(3): Pair Programming produces better code than that produced by individuals programming alone

P(3) is now explicitly relative, but still so vague as to be non-falsifiable. We have not specified what attribute/s we consider distinguish one piece of code as being "better" than another.

Suppose we think of defect density as being the measure of programmatic worth:

P(4): Pair programming produces code with a lower defect density than that produced by individuals programming alone

Now we’ve cleared up what we mean by the word "works" in P(1), let’s address another common source of vagueness – quantifiers. A quantifier is a term like "all", "some", "most" or "always". We tend to use quantifiers very casually in conversation and frequently omit them altogether. There is no explicit quantifier in P(4), so we do not know whether the claimant is proposing that the benefits of pair programming are always manifest, occasionally manifest, or just more often than not.

The quantifier chosen governs the strength of the resulting proposition. If the proposition is intended as a hard generalization (one that applies without exceptions), then a quantifier like "always" or "never" is applicable. If the proposition is intended as a soft generalization, then a quantifier like "usually" or "mostly" may be appropriate.

Suppose P(4) was actually intended as a soft generalization:

P(5): Pair programming usually produces code with a lower defect density than that produced by individuals programming alone.

P(5) nearly sounds like it could be used as a hypothesis in an empirical investigation. However the term "pair programming" is still rather vague. If we don’t clarify it, we might conduct an experiment that finds the defect density of pair programmed code to be higher than that produced by individuals programming alone, only to find that advocates of pair programming dismiss our experimental method as not being real pair programming. In other words, the definition of the term "pair programming" can be changed on an ad hoc basis to effectively render P(5) non-falsifiable.

"Pair programming" is a vague term because it carries so many secondary connotations. The primary connotations of the term are clear enough: two programmers, a shared computer, one typing while the other advises. But when we talk of pair programming we tend to assume other things that are not amongst the primary connotations. These secondary connotations need to be made explicit for the proposition to become falsifiable. To the claimant, the term "pair programming" may have the following secondary connotations:

- The pair partners contribute more or less equally, with neither one dominating the activity
- The pair partners get along with each other i.e. there is a minimum of unproductive conflict.
- The benefits of pair programming are always manifest, but to a degree that may vary with the experience and ability of the particular individuals.

To augment P(5) with all of these secondary connotations will make for a very wordy statement. At some point we have to consider what level of detail is appropriate for the context in which we are voicing the proposition.

Non-Falsifiable Propositions

Why should we seek to refine a proposition to the point that it becomes falsifiable? Because a proposition that can not be tested empirically and thereby determined true or false is beyond the scrutiny of rational thought and examination. This is precisely why such propositions are often at the heart of irrational, pseudo-scientific and metaphysical beliefs.

I contend that such beliefs have no place in the software engineering domain because they inhibit the establishment of a shared body of knowledge – one of the core features of a true profession. Instead, they promote a miscellany of personal beliefs and superstitions. In such circumstances, we cannot reliably interpret the experiences of other practitioners because their belief systems color their perception of their own experiences to an unknown extent. Our body of knowledge degrades into a collective cry of "says who?".

Here are a few examples of non-falsifiable propositions that many would consider incredible:

- There is a long-necked marine animal living in Loch Ness.
- The aliens have landed and walk amongst us perfectly disguised as humans.
- Some people can detect the presence of water under the ground through use of a forked stick.

Try as you might, you will never prove any of these propositions false. No matter how many times you fail to find any evidence in support of these propositions, it remains true that "absence of evidence is not evidence of absence." If we are willing to entertain non-falsifiable propositions such as these, then we admit the possibility of some very fanciful notions indeed.

Here a few examples of non-falsifiable propositions that many would consider credible:

- Open source software is more reliable than commercial software
- Agile techniques are the future of software development
- OO programming is better than structured programming.

These three propositions are, as they stand, just as worthless as the three propositions preceding them. The subject areas they deal with may well be fruitful areas of investigation, but you will only be able to make progress in your investigations if you refine these propositions into more specific and thereby falsifiable statements.

Engage Brain Before Engaging Flame Thrower

Vagueness and non-falsifiable propositions are the call to arms of technical holy wars. When faced with a proposition that seems set to ignite the passions of the zealots, a useful diffusing technique is to identify the non-falsifiable proposition and then seek to refine it to the point of being falsifiable. Often the resulting falsifiable proposition is not nearly as exciting or controversial as the original one, and zealots will call off the war due to lack of interest. Also, the very act of argument reconstruction can be informative for all parties to the dispute. For example:

Zealot:	*Real* programmers use Emacs
Skeptic:	How do you define a "real programmer?"
Zealot:	A real programmer is someone who is highly skilled in writing

code.

Skeptic: So what you're claiming is "people who are highly skilled in writing code use Emacs"?

Zealot: Correct.

Skeptic: Are you claiming that such people *always* use Emacs?

Zealot: Well, maybe not all the time, but if they have the choice they'll use Emacs.

Skeptic: In other words, they *prefer* to use Emacs over other text editors?

Zealot: Yep.

Skeptic: So you're claim is really "people who are highly skilled in writing code prefer Emacs over other text editors?"

Zealot: Fair enough.

Skeptic: Are you claiming that *all* highly skilled coders prefer Emacs, or could there be *some* highly skilled coders that prefer other text editors?

Zealot: I guess there might be a few weird ones who use something else, but they'd be a minority.

Skeptic: So you're claim is really "Most people who are highly skilled in writing code prefer Emacs over other text editors?”

Zealot: Yep.

Skeptic: Leaving aside the issue of how you define "highly skilled", what evidence do you have to support your proposition?

Zealot: Oh come on – *everyone* knows it's true.

Skeptic: *I* don't know it's true, so clearly not *everyone* knows it's true.

Zealot: Alright – I'm talking here about the programmers that I've worked with.

Skeptic: So are you saying that most of the highly-skilled programmers *you've* worked with preferred Emacs, or that they shared your belief that most highly-skilled programmers prefer Emacs?

Zealot: I'm talking about the editor they used, not their beliefs.

Skeptic:	So your claim is really "Of the people I've worked with, those who were highly skilled in writing code preferred to use Emacs over other text editors".
Zealot:	Yes! That's what I'm saying, for goodness sake!
Skeptic:	Not quite as dramatic as "real programmers use Emacs", is it?

You may find that it is not possible to get your opponent to formulate a specific proposition. They may simply refuse to commit to any specific claim at all. This reaction is common amongst charlatans and con men. They only speak in abstract and inscrutable terms (sometimes of their own invention), always keeping their claims vague enough to deny disproof. They discourage scrutiny of their claims, preferring to cast their vagueness as being mysterious and evidence of some deep, unspoken wisdom. If they cannot provide you with a direct answer to the question "What would it take to prove you wrong?" then you know you are dealing with a non-falsifiable proposition, and your best option may simply be to walk away.

Summary

Before engaging in any debate or investigation, ensure that the proposition being considered is at least conceivably falsifiable. A common feature of non-falsifiable propositions is vagueness.

Such propositions can be refined by:

- Defining any broad or novel terminology in the proposition
- Making implicit quantifiers explicit
- Making implicitly relative statements explicitly relative
- Making both primary and secondary connotations of the terminology explicit

[*] First published 18 Jan 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=45
[1] http://groups.yahoo.com/group/extremeprogramming

Anecdotal Evidence and Other Fairy Tales[*]

As software developers we place a lot of emphasis upon our own experiences. This is natural enough, given that we have no agreed upon body of knowledge to which we might turn to resolve disputes or inform our opinions. Nor do we have the benefit of empirical investigation and experiment to serve as the ultimate arbiter of truth, as is the case for the sciences and other branches of engineering - in part because of the infancy of Empirical Software Engineering as a field of study; in part because of the difficulty of conducting controlled experiments in our domain.

Therefore much of the time we are forced to base our conclusions about the competing technologies and practices of software development upon our own (often limited) experiences and whatever extrapolations from those experiences we feel are justified. An unfortunate consequence is that personal opinion and ill-founded conjecture are allowed to masquerade as unbiased observation and reasoned inference.

So absolute is our belief in our ability to infer the truth from experience that we are frequently told that personal experience is the primary type of evidence that we should be seeking. For example, it is a frequent retort of the XP/AM[1] crowd that one is not entitled to comment on the utility of XP/AM practices unless one has had first hand experience of them. Only then are you considered suitably qualified to make comment on the costs and benefits of the practice - otherwise "you haven’t even tried it."

Such reasoning always makes me smile, for two reasons:

1. It contains the logical fallacy called an "appeal to privileged knowledge". This is the claim that through experience one will realize some truth that forbids *a priori* description.

2. If a trial is not conducted under carefully controlled conditions, it is very likely you will achieve nothing more than a confirmation of your own preconceptions and biases.

This post is concerned with the second point. It goes to the capacity humans have to let their personal needs, prior expectations, attitudes, prejudices and biases unwittingly influence the outcomes of technology and methodology evaluations – both researchers and subjects. There are a number of statistical and psychological effects whose influence must be eliminated, or at least ameliorated, before one can draw valid deductions from human experiences. Some of these effects are briefly described in the table below. Conclusions drawn from anecdotal evidence are frequently invalid precisely because the evidence has been gathered under circumstances in which no such efforts have been made.

Observational Bias

When a researcher allows their own biases to color their interpretation of experimental results. Selective observation is a common type of observational bias in which the researcher only acknowledges those results which are consistent with their pre-formulated hypothesis.

Population Bias

When experimental subjects are chosen non-randomly and the resulting population exhibits some unanticipated characteristic that is an artifact of the selection process, which influences the outcome of an experiment in which they participate.

The Hawthorne Effect

Describes the tendency for subjects to behave uncharacteristically under experimental conditions where they know they are being watched. Typically this means the subjects improve their performance in some task, in an attempt (deliberate or otherwise) to favorably influence the outcome of the experiment.

The Placebo Effect

Describes the tendency of strong expectations, particularly among highly suggestible subjects, to bring about the outcome expected through purely psychological means.

Logical Fallacies

Conclusions drawn from anecdotal evidence often exhibit one or more of the following deductive errors:

Post Hoc Ergo Propter Hoc

Meaning "after this, therefore because of this". When events A and B are observed in rapid succession, the post hoc fallacy is the incorrect conclusion that A has caused B. It may be that A and B are correlated, but not necessarily in a causal manner.

Ignoring Rival Causes

To disregard alternative explanations for a particular effect, instead focusing only upon a favorite hypothesis of the researcher. It is common to look for a simple cause of an event when it is really the result of a combination on many contributory causes.

Hasty Generalization

The unwarranted extrapolation from limited experimentation into a broader context.

Examples

The following scenarios demonstrate how easily one or more of the above factors can invalidate the conclusions that we reach based on our own experience - thereby reducing the credibility of those experiences when later offered as anecdotal evidence in support of a conclusion.

The Linux Enthusiast

Chris is a Linux enthusiast. On his home PC he uses Linux exclusively, and often spends hours happily toying with device drivers and kernel patches in an effort to get new pieces of hardware working with his machine. In his work as a software developer he is frequently forced to use Microsoft Windows, which he has a very low opinion of. He is prone to waxing lyrical on the unreliability and insecurity of Windows, and the evil corporate tactics of Microsoft. Whenever he experiences a Blue Screen of Death on his work machine, his cubicle neighbors know that once the cursing subsides they are in for another of his speeches about the massive productivity hit that Windows imposes on the corporate developer. When surfing the web during his lunch hours, if he should come across a reference to Linux being used successfully as an alternative to Windows, then he will print out the article and file it away for future reference. He is confident that it is only a matter of time before Linux replaces Windows on the desktop, both in business and at home.

Analysis: Chris exhibits *observational bias* in a few ways. The hours he spends getting his Linux machine to recognize a new piece of hardware is enjoyable to him, and so he chooses not to observe that the same outcome might be achieved on a Windows system in a minute, thanks to plug-and-play. When he gets a BSOD, he chooses to observe its negative effect on his productivity while he waits for a reboot, but chooses to disregard the productivity cost of his subsequent anti-Microsoft pontifications. When surfing the web, he *selectively observes* those stories which are pro-Linux and/or anti-Microsoft in nature. Indeed, the media is complicit in this practice, because such stories make good press. There may be many more occasions in which Linux was unsuccessful in usurping Windows, but they are unremarkable and unlikely to attract media coverage. His confidence in Linux's ultimate victory based upon his selective observations is a very *hasty generalization*.

The XP Proponent

Ryan and his team have been reading a lot about XP recently and are keen to try it out on one of their own projects. They have had difficulty getting permission to do so from their management, who are troubled by some aspects of XP such as pair programming and the informal approach to documentation. Through constant badgering, Ryan finally gets permission to use XP on a new project. But he is warned by his management that they will be watching the project's progress very carefully and reserve the right to switch the project over to the company's standard methodology if they think XP is not working out. Overjoyed, Ryan's team begins the new project under XP. They work like demons for the next six months, doing everything in their power to make the project a success. At the end of that time, the project delivers a high quality first release into the hands of a few carefully chosen customers. Feedback from these customers is unanimously positive. Management is suitably impressed. Ryan and his team breathe a sigh of relief.

Analysis: The participants are a self-selected group of enthusiasts, which is an obvious source of *population bias*. It could be that they have an above-average level of ability in their work, and a commensurately higher level of enthusiasm and dedication - which drives them to try new approaches like XP. Their project's success may be partly or entirely attributable to these greater capabilities they already had. Knowing they are being closely evaluated by management and have put their necks on the line by trying XP despite management's concerns, they are also victims of the *Hawthorne Effect*. They are very motivated to succeed, because they perceive potential adverse consequences for themselves individually if they should fail. If Ryan's team or their management attributes the project's success to XP itself, then they are guilty of *ignoring the rival causes* just described. It may be that they succeeded *despite* XP, rather than because of it.

The Revolutionary

Seymour thinks there is something wrong with the way university computing students are taught to program. He feels there is insufficient exposure to the sorts of problems and working conditions they will encounter when they finish their degrees. He strongly believes that students would become better programmers and better employees if there were a greater emphasis upon group programming assignments in the academic environment. This would enable them to develop the skills necessary to function effectively in a team, which is the context in which they will spend most of their working lives. To demonstrate the effectiveness of the group approach, he asks for some volunteers from his third year software engineering class to participate in an experiment. Rather than do the normal lab work for their course, which is focused on assignments to be completed by the individual, they will do different labs designed to be undertaken in groups of four or five. These labs will be conducted by Seymour himself. About 30 students volunteer to take part. At the end of the semester, these students sit the same exams as the other students. Their average mark is 82% while the average mark of the other students is 71%. Seymour feels vindicated and the volunteer students are pleased to have taken part in a landmark experiment in the history of computing education.

Analysis: Here is a case of *population bias* that any competent researcher would be ashamed of. The volunteer group is self-selected, and so may be biased toward those students that are both more interested and more capable. Poor performing, disinterested students would be unlikely to volunteer. The *Hawthorne Effect* comes into play due to the extra focus that Seymour places upon his volunteer group. They may receive extra attention and instruction as part of their labs, which may be enough in itself to improve their final grades. Additionally, knowing they are part of a select group, at some level they will be motivated to please the researcher and demonstrate that they have performed well in their role as "lab rats." Their superior performance in the final exam may be a result of these confounding factors, and have nothing to do with the difference between individual and group instruction. It would certainly be a *hasty generalization* to conclude that their better exam results will translate into better performance in the workforce.

Conclusion

I hope this post will give you pause for thought when you next conduct a technology trial, and when you are next evaluating anecdotal evidence supplied to you by friends and colleagues. Because personal experiences are particularly vivid, we often tend to over-value them. From there, we can easily make unwarranted generalizations and overlook the confounding effect of our own preconceptions and biases.

In particular, next time one of the XP/AM crowd voice the familiar retort of "How could you know? You haven't even tried it" - bear in mind that in the absence of quantification and controlled experimental technique, *they* don't know either.

[*] First published 22 Mar 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=49

[1] Extreme Programming / Agile Methods

Function Points: Numerology for Software Developers*

"Where else can one get such a marvelous return in conjecture from such a modest investment of fact?" – Mark Twain

Numerology is the study of the occult meanings of numbers and their influence on human life[1]. Numerologists specialize in finding numeric relationships between otherwise disparate figures, and attributing to them some greater significance.

For instance, some claim that by adding up the component numbers in your birth date, together with the numeric equivalent of your name (where A=1, B=2 etc) then a figure is derived that, if properly interpreted, can yield insight into your personality.[1]

Others consider that the reoccurrence of the number 19 in Islamic texts is evidence of their authorship by a higher being [2]. The Koran has 114 (6 x 19) chapters and 6346 verses (19 x 334) and 329,156 (19 x 17,324) letters. The word "Allah" appears 2,698 (19 x 142) times. The sum of the verse numbers that mention Allah is 118,123 (19 x 6,217).

Pyramids are a favorite topic for numerologists, and there are dozens of "meaningful" numeric relationships to be found in their dimensions. For instance, the base perimeter of the Great Pyramid of Cheops is 36,515 inches – 100 times the number of days in the solar year. And so on.

We can laugh at such desperate searches for meaning, but before we laugh too hard we should consider that software development has its own brand of numerology, which we have given the grand name of Function Point Analysis (FPA).

Overview Of Function Points

FPs were proposed in 1979 as a way of finding the size of a piece of software given only its functional specification. It was intended that the FP count of an application would be independent of the technology, people and methods eventually used to implement the application, focusing as it did upon the functionality the application provided to the user. Broadly speaking, basic FPs are calculated by following these steps:

1. Divide a functional view of the system into components.

2. Classify each component as being one of five types – external input, external output, external inquiry, internal logical file or external interface file.
3. Classify the complexity of each component as low, average or high. The rules for performing this classification vary by component type.
4. For each type of component, multiply the number of components of that type by a numeric equivalent of the complexity e.g. low = 3, average = 4, high = 6. The numeric equivalents that apply vary by component type.
5. Sum the results of step 4 across all five component types. The total is a figure called Unadjusted Function Point count (UFP).You can then multiply the UFP by a Value Adjustment Factor (VAF) which is based on consideration of 14 general system characteristics, to yield the final Function Point count.

I won't bore you with the excruciating specifics of the component calculations. The above gives you some idea of the nature of FP counting and it's reliance upon subjective judgments. Specifically, the placement of component boundaries and the values chosen for the many weighting factors and characteristics are all determined on a subjective basis. Some of that subjectivity has been embodied in the standardized FP counting rules that are issued by the International Function Point Users Group (IFPUG).[3]

So lacking have FPs been found, that there has been a steady stream of proposed improvements and alternatives to them since 1979. But none of these have challenged the basic FP ethos of modeling functional size as a weighted sum of arbitrarily selected attributes. They simply change the number and definition of those attributes, and the means by which they are mangled together into a final figure. The basic chronology of the FP family tree has been:

1979	Function Points (Albrecht)
1986	Feature Points (Jones)
1988	Mark II Function Points (Symons)
1989	Data Points (Sneed)
1991	3 D Function Points (Boeing)
1994	Object Points (Sneed)
1997	Full Function Points (St. Pierre et. al)
1999	COSMIC Full Function Points (IFPUG)

To understand why the FP and its many variants are fundamentally flawed, it is first necessary to understand the difference between *measuring* and *rating*.

Measurement Vs. Rating

To *measure* an attribute of something is to assign numbers to it on an objective and empirical basis, so that the relationships between the numbers preserve any intuitive notions and empirical observations about that attribute.[4]

For example, the metric meter is a measure, which implies:

- 4 meters is twice as long as 2 meters, because 4 is twice 2
- The difference between 9 and 10 meters is the same as the difference between 1 and 2 meters, because 10-9 = 2-1
- If you moved 4 meters in 2 seconds (at constant velocity) then you moved 2 meters in the first second and 2 meters in the last second.
- If two different people measure the same length to the nearest meter, they will get the same number.

To *rate* an attribute of something is to assign numbers to it on a subjective and intuitive basis. The relationships between the numbers do *not* preserve the intuitive and empirical observations about the attribute. In contrast to the above example, consider the rating out of 10 that a reviewer gives a movie:

- A movie that gets a 4 is not twice as good as a movie that gets a 2.
- The difference between movies that get 9 and 10 is not the same as the difference between movies that get 1 and 2.
- A 2 hour movie that gets a 6 did not rate 3 for the first hour and 3 for the second hour.
- Two different people rating the same movie may award different ratings.

To clarify, suppose a reviewer expresses their assessment of a movie in words rather than numbers. Instead of rating a movie from 1 □ 10, they rate it from "abysmal" to "magnificent". We might be tempted to think a movie that gets an 8 is twice as good as a movie that gets a 4, but we would surely not conclude that "very good" is twice as good as "disappointing". We can express a rating using any symbols we want, but just because we choose numbers for our symbols does not mean that we confer the properties of those numbers upon the attribute we are rating.

In summary:

- A *measurement* is objective and can be manipulated mathematically.
- A *rating* is subjective and cannot be manipulated mathematically.

Function Points Are A Rating, Not A Measurement

From the above, it is clear that FPs are a rating and not a measurement, due to the subjective manner in which they are derived. Hence, they cannot be manipulated mathematically. And yet the software literature is rife with examples of researchers attempting to do just that. Many researchers and reviewers continue to ignore the fundamental implications of the non-mathematical nature of the FP[5], such as:

- *You cannot measure productivity using FPs* – If a team completes an application of 250 FP in 10 weeks, their productivity is not 25 FP/week. The figure "25" has no meaning. Similarly, a given team need not take 50% longer to write a 1800 FP application as they will a 1200 FP application.
- *You cannot compare FP counts numerically* – An application of 1000 FP is not twice as big, complex or functional as an application of 500 FP. The first application is not "twice" the second in any meaningful sense.
- Y*ou cannot compare FPs from disparate sources* – The subjectivity of FP analysis makes it sensitive to contextual variations in application domain, technology, organization and counting method.

Given such limitations, there are very few valid uses of an application's FP count. If the FP counts of two applications differ markedly, and their contexts are sufficiently similar, then you *may* be justified in saying that one is functionally bigger than the other, but not by how much.[3] The notion that FPs can participate in mathematical calculations, and thereby be used for scheduling, effort and productivity measures, is without theoretical or empirical basis.

Why Are Function Points So Popular?

- Although their use may have declined in recent years, Function Points are still quite popular. There are several factors which might account for their continued usage, despite their essential invalidity:
- The fact that other organizations use FPs is enough to encourage some to follow suit. However, we should be aware that an *argument from popularity* has no logical basis. There are many beliefs that are both widely held and false. The popularity of FPs may only be indicative of how desperately the industry would like there to be a single measure of functional size that can be calculated at the specification stage. It certainly would be desirable for such a measure to exist, but we cannot wish such a metric into existence, no matter how many others have the same wish.
- Some researchers claim to have validated function points (in their original form, or some later variant thereof). However, if you examine the details of these experiments, what you will find is pseudo-science, ignorance of basic measurement theory and statistics, and much evidence of "fishing for results." There is a lot of fitting of models to historical data, but not a lot of using those models to predict future data. This is not so surprising, for the general standard of experimentation in software is very poor, as Fenton observes. Altman makes an observation[6] about the legion of errors that occur in medical experimentation that could apply equally well to software development:
- "The main reason for the plethora of statistical errors is that the majority of statistical analyses are performed by people with an inadequate understanding of statistical methods. They are then peer reviewed by people who are generally no more knowledgeable."
- Hope springs eternal. Rather than concede that efforts to embody functional size in a single number are misguided, it is consoling to think that FPs are "nearly there", just a few more tweaks away from being useful. Hence the many FP variants that have sprung up.
- FP enthusiasts selectively quote the "research" that is in their favor, and ignore the rest. For example, the variance between FP counts determined by different analysts is often quoted as "plus or minus 11 percent."[7] However other sources[8] have reported worse figures, such

as a 30% variation *within* an organization, rising to more than 30% *across* organizations.

- Some choose to dismiss the theoretical invalidities of FPs as irrelevant to their practical worth. Their excuses may have some appeal to the average developer, but don't withstand scrutiny. Examples of such excuses are:

    - *As long as FPs work, who cares what basis they have or don't have?* - The problem is that in general, FPs *don't* work. Even FP adherents will admit to the numerous shortcomings of FPs, and the need to constrain large numbers of contextual factors when applying them. Witness the various mutations of FP that have arisen, each attempting to address some subset of the numerous failings of FPs.

    - *It doesn't matter if you're wrong, as long as you're wrong consistently*[9] – Unfortunately, unless you know *why* you're wrong, you have no way of knowing if you are indeed being *consistently* wrong. FPs are sensitive to a great many contextual factors. Unless you know what they are and the precise way they effect the resulting FP count, you have no way of knowing the extent to which your results have been influenced by those factors, let alone whether that influence has been consistent.

Function Point's True Believers

FPs have attracted their own league of True Believers – like many technical schools whose tenets, lacking an empirical basis, can only be defended by the emotional invective of their adherents. I encountered one such adherent recently in David Anderson, author of "Agile Project Management." Anderson made some rather pompous observations[10] on his blog as to how surprising it was that people should express disbelief regarding his claims to 5 and 10-fold increases in productivity using TDD, AM and (insert favorite acronym here)FDD. I replied that their incredulity might stem from the boldness of his claims or the means by which he collected his data, rather than an inherently obstreperous attitude. He indicated his productivity data was expressed in FPs per unit time! I tried explaining to him that FPs cannot be used to measure productivity, because not all FPs are created equal, as explained above. He wasn't interested.

That discussion has now been deleted from his blog. He also denied me permission to reproduce that portion of it which occurred in email.

Such is the attitude I typically encounter when dealing with self-styled gurus and experts. There is much talk of science and data, but as soon as you express doubt regarding their claims, there is a quick resort to insult and posture. Ironic, given that doubt and criticism are the basic mechanisms that give science the credibility that such charlatans seek to cloak themselves in.

Why Must Functional Size Be A Single Number?

The appeal, and hence the popularity, of FPs is their reduction of the complex notion of software functional size to a single number. The simplicity is attractive. But what basis is there for believing that such a single-figure expression of functional size is even possible?

Consider this analogy. When you walk into a clothing store, you characterize your size using several different measures. One figure for shirt size, another for trouser size, another for shoe size and another for hat size. What if, by way of misguided reductionism, we were to try and concoct a single measure of clothing size and call it *Clothing Points*. We could develop all sorts of rules and regulations for counting Clothing Points, including weighting factors accounting for age, diet, race, gender, disease and so on. We might even find that if we sufficiently controlled the influence of external factors, given the limited variations of the human form, we might eventually be able to find some limited context in which Clothing Points were a semi-reasonable assessment of the size of all items of clothing. We could then walk into a clothing store and say "My size is 187 Clothing Points" and get a size 187 shirt, size 187 trousers, size 187 shoes and size 187 hat. The items might even fit, although we would likely sacrifice some comfort for the expediency and convenience of having reduced four dimensions down to a single dimensionless number.

The search for a grand unified "measure" of functional size may be just as foolhardy as the quest for uni-metric clothing.

Conclusion

The continued use and acceptance of Function Point Analysis in software development should be a source of acute embarrassment to us all. It is a prime example of muddle-headed, pseudo-scientific thinking, that has persisted only because of the general ignorance of measurement theory and valid experimental methodology that exists in the development community. We need to stop fabricating and embellishing arbitrary sets of counting rules. In doing so, we are treating these formulae as if they were incantations whose magic can only manifest when precisely the correct wording has been discovered, but whose inner workings must forever remain a mystery. Rather, we need to go back to basics and work towards understanding the fundamental technical dimensions that contribute to the many and varied notions of an application's functional size. How can we hope to measure something when we can’t even precisely define what that something is? Empiricism holds some promise as a means to improve software development practices, but the pseudo-empiricism of Function Point Analysis is little more than numerological voodoo.

---

[*] First published 28 Jun 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=59

[1] *The Skeptic’s Dictionary*, R. Carroll, Wiley and Sons, 2003. http://www.skepdic.com/

[2] *Did Adam and Eve Have Navels?*, M. Gardner, W.W. Norton and Company, 2000

[3] http://www.ifpug.org/

[4] *Software Measurement: A Necessary Scientific Basis*, N. Fenton, IEEE Trans. Software Eng., Vol. 20, No. 3, 1994

[5] *The Problem with Function Points*, B. Kitchenhas, IEEE Software, March/April 1997

[6] *Statistical Guidelines for Contributors to Medical Journals*, Altman, Gore, Gardner, Pocock, British Medical Journal, Vol. 286, 1983

[7] *Why We Should Use Function Points*, S Furey, IEEE Software, March/April 1997

[8] *Comparison of Function Point Counting Techniques*, J.Jeffery, G. Low, M. Barnes, IEEE Trans. Software Eng., Vol. 19, No. 5, 1993

[9] *Measurement and Estimation*, Burris

[10] http://www.agilemanagement.net/Articles/Weblog/WorldClassVelocity.html

Programming and the Scientific Method*

In 1985 Peter Naur wrote a rather cryptic piece entitled *Programming as Theory Building*[1] in which he drew an analogy between software development and the scientific method. Since then, other authors have attempted to co-opt this analogy as a means of enhancing the perceived credibility of particular programming practices. This post aims to explain the analogy between the scientific method and programming, and to explore the limitations of that analogy.

The Scientific Method

There is no canonical representation of the scientific method. Different sources will explain it in different ways, but they are all referring to the same logical process. For the purposes of this discussion, I will adopt a simplified definition of the scientific method, considering it to be comprised of the following activities repeated in a cyclic manner:

1. *Model* – Form a simplified model of a system by drawing general conclusions from existing data.
2. *Predict* – Use the simplified model to make a specific prediction about how the system will behave when subject to particular conditions.
3. *Test* – Test the prediction by conducting an experiment.

If the test confirms our prediction, we return to step 2 and make a new prediction based upon the same model. Otherwise, we return to step 1 and revise our model so that it accounts for the results of our most recent test (and all preceding tests).

More formal descriptions of the scientific method often include the following terms:

Hypothesis – A testable statement accounting for a set of observations. It is equivalent to the *model* in the above description.

Theory – A well supported and well tested hypothesis or set of hypotheses.

Fact – A conclusion confirmed to such an extent that it would be reasonable to offer provisional agreement.[2]

An Example Of The Scientific Method

Suppose you are given a sealed black box that has only three external features – two toggle switches marked A and B, and a small lamp. By playing around with the switches you notice that certain combinations of switch positions result in the lamp lighting up. Your task is to use the scientific method to develop a theory of how the box operates. In other words, to create a model which can account for the observed behavior of the box.

Round 1

Model: Casual observation suggests that the switches and lamp are connected in circuit with an internal power source. Let's suppose that this is the case, and that the two toggle switches are wired in series.

Predict: If our model is accurate, then we should find that turning both switches on causes the lamp to light up.

Test: We get the box, turn both switches on and find that the lamp does indeed light up. Our model has been partially verified. But there are other predictions we can make based upon it.

Round 2

Model: As in experiment 1.

Predict: If our model is accurate, then we should find that turning switch A off and switch B on causes the lamp to go out.

Test: We get the box, turn switch A off and switch B on and find that the lamp actually lights up. Our prediction was incorrect, therefore our model is wrong

Round 3

Model: Now we need to rework our model so that it correctly accounts for all our observations thus far. Then we can use it as a basis for further prediction. Suppose the box were wired with the two toggle switches in parallel. That would account for our observations from rounds 1 and 2. Let's make that our new model.

Predict: If this new model is accurate, then we should find that turning switch A on and switch B off causes the lamp to light up.

Test: We get the box, turn switch A on and switch B off and find that the lamp actually goes off. Our prediction was incorrect; therefore our new model is wrong.

Round 4

Model: Once again, we need to reformulate our model so that correctly accounts for all of our existing observations. After some thought, we realize that if the box were wired so that only switch B effected the lamp, with switch A out of the circuit entirely, then this would account for all of our existing observations, as well as giving us a new prediction to test.

Predict: If this latest hypothesis is true, then we should find that turning switch A off and switch B off causes the lamp to go out.

Test: We get the box, turn switch A off and switch B off and observe that the lamp does indeed go out. Our prediction was correct, and our model is consistent with our observations from all four experiments

You can see why the scientific method is sometimes described as being very inefficient – there is a lot of trial and error involved. But it's important to note that it's not random trial and error. If we just made random predictions and then tested them through experiment, all we would end up with is a disjoint set of cause/effect observations. We would have no way of using them to predict how the system would behave under situations that we hadn't already deserved. Instead, we choose our predictions deliberately, guided by the intent of testing a particular aspect of the model currently being considered. In this way, each experiment either goes some way toward confirming the model, or confuting it.

Note that all we can *ever* have is a model of the system. We make no pretense to know the truth about the system in any absolute sense. Our model is simply *useful*, at least until new observations are made that our model can't account for. Then we must change it to accommodate the new observations. This is why all knowledge in science (even that referred to as fact) is actually provisional and continually open to challenge.

A Programming Example

The following example demonstrates how software development is similar to the scientific method.

The task is to develop an application which models the behavior of the black box in the above example. The software will present a simple GUI with two toggle buttons marked A and B, and an icon which can adopt the appearance of a lamp turned on or off. The lamp icon should appear to be turned on as if the lamp were a real lamp connected to an internal power source, and the toggle buttons were toggle switches, with switch B in circuit with the lamp, and switch A out of circuit.

The table below compares the activities in the scientific method with their programming counterparts. Keep these analogs in mind as you read through the following example.

	Scientific Method	**Programming**
Model	Form a simplified model of a system by drawing general conclusions from existing data	Developing a mental model of how the software works
Predict	Use the simplified model to make a specific prediction about how the system will behave when subject to particular conditions.	Taking a particular case of interaction with that model, and predicting how the software will respond
Test	Test the prediction by conducting an experiment.	Subjecting software to a test and getting a result.

Round 1

Model: Unlike experimentation, we begin by assuming our model is correct. It is created from our requirements definition and states "The lamp icon should appear to be turned on as if the lamp were a real lamp connected to an internal power source, and the toggle buttons were toggle switches, with switch B in circuit with the lamp, and switch A out of circuit."

Predict: If the software is behaving correctly, toggling both buttons on should result in the lamp icon going on.

Test: We run the software, toggling the buttons A and B on, and observe that the lamp icon does indeed come on. So far our hypothesis has been confirmed; which is to say, the software behaves as the requirements say it should. But there are other behaviors specified by the requirements

Round 2

Model: As per round 1

Predict: If the software is behaving correctly, then toggling button A off and button B on will cause the lamp icon to go on.

Test: We run the software, toggle button A off and button B on, and find that the lamp icon actually turns off. Our prediction was incorrect; therefore our software is not behaving as per its requirements. Instead of adjusting our model to suit the software, we adjust the software to suit the model i.e. we debug the software. In the software world, we can change the "reality" we are observing to behave however we want - unlike the real world where we have to adjust our model to fit an invariant reality. Once the software behaves in a manner consistent with the above prediction, we have to repeat our test from round 1 (i.e. regression test), to confirm that the prediction made there still holds i.e. that we haven’t "broken" the software reality.

Round 3

Model: As per round 1.

Predict: If the software is behaving correctly, then toggling button A on and button B off should cause the lamp icon to turn off.

Test: We run the software, toggle button A on and button B off and find that the lamp icon actually turns on. Our prediction was incorrect; therefore our software is in error. Once again we debug the software until it behaves in a manner consistent with the above prediction. Then we regression test by repeating the tests in rounds 2 and 3.

Round 4

Model: As per round 1.

Predict: If the software is behaving correctly, then toggling buttons A and B off should cause the lamp icon to turn off.

Test: We run the software, toggle buttons A and B off and find that the lamp icon does indeed turn off. Our prediction was correct; therefore the software is behaving as per its requirements.

Notice the critical difference between programming and experimentation. In experimentation, reality is held invariant and we adjust our model until the two are consistent. In programming, the model is held invariant and we adjust our reality (the software) until the two are consistent.

Limits Of The Analogy

Rote performance of the model/predict/test cycle does not mean that one is doing science, or even that one's activities are science-like. There are critical attributes of the way these activities are carried out that must be met before the results have scientific validity. Two of these are objectivity and reproducibility. Some authors have taken the analogy between scientific method and programming too far by neglecting these attributes.

McCay[3] contends that pair programming is analogous to the peer review process that scientific results undergo before being published. The reviewers of a scientific paper are chosen so that they are entirely independent of the material being reviewed, and can perform an objective review. They must have no vested interest in the material itself, and no relationship to the researcher or anyone else involved in the conduct of the experiment. To this end, scientific peer reviews are often conducted anonymously. Clearly this independence is missing in pair programming. Both parties have been intimately involved in the production of the material being reviewed, and as a coauthor each has a clear personal investment in it. They have participated in the thought processes that lead to the code being developed, and so can no longer analyze the material in an intellectually independent manner.

Mugridge [3] contends that the continuous running of a suite of regression tests is equivalent to the concept of scientific reproducibility. But here again, the independence is missing. A single researcher arriving at a particular result is not enough for those results to be considered credible by the scientific community. Independent researchers must successfully replicate these results, as a way of confirming that they weren't just a chance occurrence, or an unintentional byproduct of situational factors. But running regression tests does not provide such confirmation, because each run of the regression tests is conducted under exactly the same circumstances as the preceding ones. The same tests are executed in the same environment over and over again, so there is no independence between one execution and the next. Thus the confirming effect of scientific reproducibility is lost.

Both Mugridge and McCay try and equate the XP maxim "do the simplest thing that could possibly work" (DTSTTCPW) with Occam's Razor. Occam's razor is a principle applied to hypothesis selection that says "Other things being equal, the best hypothesis is the simplest one, that is, the one that makes the fewest assumptions." Because the scientific hypothesis is analogous to the *system metaphor* in XP, the XP equivalent of Occam's Razor would be "Other things being equal, the best system metaphor is the simplest one, that is, the one that makes the fewest assumptions." However XPers often invoke DTSTTCPW with regard to implementation decisions, not choice of metaphor. Indeed, the metaphor is one of the least used of XP practices.[4]

Additionally, the "all other things being equal" part of Occam's razor is vital, and neglected in XP's DTSTTCPW slogan. We evaluate competing hypotheses with respect to the criteria of adequacy [5] – which provide a basis for assessing how well each hypothesis increases our understanding. The criteria include testability, fruitfulness, simplicity and scope. Note that simplicity is only one of the factors to consider. The scope of a hypothesis refers to its explanatory power; how much of reality it can explain and predict. We have a preference for a hypothesis of broader scope, because it accounts for more natural phenomena. In a programming context, suppose we have two competing models of a piece of software's operation. One is more complex than the other, but the more complex one also has greater scope. Which one is better? It's a subjective decision; but it should be clear that considering simplicity alone is a naive basis for hypothesis selection.

Conclusion

OK, so there are parallels between the scientific method and programming. Aside from the intellectual interest, what value is there in recognizing these parallels?

Naur claims that the theory of a piece of software corresponds to the model that the programmer builds up in their head of how it works. Such a theory might say "The software is like a box with two toggle buttons and a lamp", or "The software is like an assembly line with pieces being added on as the item proceeds". Perhaps multiple metaphors are used. Once a programmer has a theory (model) of the software in their head, they can talk about and explain its behavior to others. When they make changes to the code, they do so in a way that is consistent with the theory and therefore "fits in" with the existing code base well. A programmer not guided by such a theory is liable to make modifications and extensions to the code that appear to be "tacked on" as an afterthought, and not consistent with the design philosophy of the existing code base. I believe there is some validity in this notion.

Cockburn then extends this by claiming that this theory is what is essential to communicate (in documentation or otherwise) from one generation of programmers to the next: "What should you put into the documentation? That which helps the next programmer build an adequate theory of the program". He also sees this as validation of the "System Metaphor" practice from XP. Perhaps so, but I think there is only limited utility in identifying *what* has to be communicated. The real problem is identifying how to communicate; how to persist that knowledge in a robust form, and transfer it from one programmer to another as new programmers arrive on a project and old ones leave.

---

[*] First published 21 Aug 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=64
[1] *Programming as Theory Building*, Peter Naur
[2] *Why People Believe Weird Things*, Michael Shermer
[3] *if (extremeProgramming.equals(scientificMethod))*, Larry McCay
[4] *Agile and Iterative Development*, C. Larman
[5] *How to Think About Weird Things*, 3rd edition, T. Schik and L. Vaughn, McGraw Hill, 2002

From Tulip Mania to Dot Com Mania*

“Those who cannot remember the past are condemned to repeat it.” – George Santayana

Those of us working in IT tend to think of ourselves as being modern, savvy and much more advanced than our forebears. This conviction is often accompanied by a certain degree of hubris, and a somewhat derisive attitude towards older technologies and practitioners. You’ve probably encountered this ageist bias in your own work place, or even displayed it yourself. Older members of our profession are viewed as out-dated and irrelevant. Older programming languages such as C and FORTRAN are viewed as inherently inferior to those more recently introduced such as Java and C#. Contempt for that which has come before us is as common place as the fascination with novelty and invention that breeds it.

In our struggle to stay abreast of the rapid rate of change in our industry, our focus is so intensely upon the present and immediate future, that we neglect the lessons of the past. We excuse our parochialism by kidding ourselves that the pace of technological makes any comparison with the past all but irrelevant anyway. But here lies a serious error in thinking – for although technology changes rapidly, people do not. For example, throughout history there are numerous examples of large groups of people succumbing to mass panics, group delusions and popular myths. Notable events are:

- The *Martian Panic* of 1938, in which many Americans became convinced that a radio broadcast of H.G. Well’s War of the Worlds was a news broadcast of an actual Martian invasion, leading some to flee their homes to escape the alien terror.[1]
- The *Roswell Flying Saucer* crash of 1947, a myth sustained by many even today.
- The widespread belief in *Satanic Ritual Abuse* of children in America in the 1970’s and 1980’s.
- The *Witch Mania* of the 15th-17th centuries on multiple continents. Exemplified by the Salem witch trials of 1692.

- The *Face on Mars* myth of 1976

It is easy to dismiss such phenomena as unique to their times, the like of which could never be experienced by modern, technology-aware, scientifically informed people such as ourselves. But we view our modern world with old brains. Psychologically, we have the same predilections and foibles as the witch-hunters and alchemists of centuries past. We still experience greed, we still feel a need to belong to a group, and we can still sustain false and irrational beliefs if we see others doing the same.

To illustrate our continuing susceptibility to irrational group behaviors, consider the Tulip Mania of the 1630s, which exhibits striking parallels with the dot-com mania that would follow it some 400 years later.

Tulip Mania

The collecting of tulips began as a fashion amongst the wealthy in Holland and Amsterdam in the late 16th century[2]. The popularity of the flower spread to England in 1600, and filtered down from the upper class to the middle class. By 1635 the mania had reached its peak amongst the Dutch, and preposterous sums were being paid for bulbs of the rarer varieties. A single bulb of the species Admiral Liefken sold for 4400 florins, and a Semper Augustus for 5500 florins, at a time when a sheep cost 10 florins.

In 1636 the demand for rare tulips became so great that regular marts for their sale were established on the Stock Exchange of Amsterdam. At this time, speculation in tulip bulbs appeared, and those fortunate enough to buy low and sell high quickly grew rich. Seeing their friends and colleagues profiting from the tulip mania, ordinary citizens began converting their property into cash and investing in bulbs. All were convinced that Europe's current infatuation with tulips would continue unabated for the foreseeable future and that vast wealth awaited those who could satiate the frenzied demands that were sure to come from the rest of Europe.

But the more prudent began to see that this artificial price inflation could not be sustained for much longer. As confidence dropped, so too did the market price of tulips – never to rise again. Those caught with significant investments in bulbs were ruined, and Holland's economy suffered a blow from which it took many years to recover.

There are obvious similarities with the dot com boom – the artificial escalation of value, the widening scope of investors, the confusion of popularity with substance, the progression from investor over-confidence to widely held belief, and finally, the sudden deflation of value promoted by the growing awareness of the role that non-financial factors were playing in the trend.

Conclusion

It has always been the province of recent generations to view the mistakes of earlier generations with a contempt derived from the assumption that they are somehow immune to such follies. Those of us who are more technology-aware than some others are particularly prone to this. And yet, even the geekiest techno-junkie can fall prey to the same psychological and sociological traps that have plagued our species for centuries. Indeed, far from inuring us to metaphysical thinking, it seems that the sheer success of science has lead many to deliberately pursue "alternative" beliefs as a way of restoring some feeling of mystery and wonder into their lives. A 1990 Gallup poll of 1236 adult Americans found that 52% believed in astrology, 46% in ESP and 19% in witches.[3] The result is that superstition and technology are both coexistent and symbiotic. As software developers, we need to heed the lessons of the mass manias of the past, acknowledge that we are still psychologically vulnerable to them today, and guard against their re-emergence by making a deliberate effort to think critically about the trends, fashions and hype which so predominate our industry.

---

[*] First published 5 Jun 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=56
[1] Hoaxes, Myths and Manias, R. Bartholomew and B. Radford, Prometheus Books, 2003
[2] *Extraordinary Populat Delusions and The Madness of Crowds*, Charles Mackay, Wordsworth Editions, 1995
[3] *Why People Believe Weird Things*, Michael Shermer, Henry Holt and Company, 2002

The Industry

The Crooked Timber of Software Development*

"Out of the crooked timber of humanity no straight thing was ever made." – Immanuel Kant

Imagine you are a surgeon. You are stitching a wound closed at the end of a major procedure, when you are approached by the chief surgeon, clad in theatre garb. He explains that, inkeeping with recently introduced hospital policy, you are required to use a cheaper, generic brand of suture material, rather than the more common (and more expensive) brand you are accustomed to using. He orders you to undo the stitching you've done, and redo it using the generic brand.

Now you are in an ethical quandary. You know that the cheaper suture material is not of the same strength and quality as the usual type. You also know that it is a false economy to skimp on sutures, given that the amount of money to be saved is trivial, but the increased risk to the patient is decidedly non-trivial. Further, it seems unconscionable to be miserly on such critical materials. But on the other hand, the chief surgeon wields a lot of political might in the hospital, and it would no doubt be a career-limiting move to ignore his instruction. So what do you do?

As a health professional, there is simply no question. You are legally and ethically obliged to act in the best interests of the patient and there are serious consequences if you fail to do so. The penalties for malpractice include financial, legal and professional remedies. You can be fined, sued for malpractice, or struck from the register and rendered unable to practice. In the light of the system's support and enforcement of good medical practice, you complete the stitching using the standard suture material, then express your concerns to the chief surgeon. If you don't get satisfaction, you can take the matter further.

Now let's examine a similar situation in our own industry. Suppose a software developer is trying to decide which of a set of competing technologies should be used on a project. One technology stands out as clearly superior to the others in terms of its suitability to the project's circumstances. Upon hearing of the technology chosen, the company's senior architect informs the developer that they have made the wrong decision, although they cannot explain why that is the case. The architect directs you to use a technology you know to be inferior, and makes it clear that it would be a career-limiting move to ignore his instruction. Again, what do you do?

My observations over the last twelve years working as a software developer leave me in no doubt what the probable outcome is. You shake your head in disbelief, and use the technology you are instructed to use, knowing that the best interests of both the project and its sponsors has just been seriously compromised. Why is the situation so different from the previous medical scenario? The basic answer is this: medicine is a profession, but software development merely an occupation.

A Profession Is More Than An Occupation

As it is used in common parlance, the word "profession" refers to the principle occupation by which you earn an income. But this is not its true meaning. A true profession has at least the following characteristics:[1]

- *Minimum educational requirements* – Typically an accredited university degree must be completed.
- *Certification / licensing* – Exams are taken to ensure that a minimum level of knowledge has been obtained. These exams target an agreed upon body of knowledge that is considered central to the profession.
- *Legally binding code of ethics* – Identifies the behaviors and conduct considered appropriate and acceptable. Failure to observe the code of ethics can result in ejection from professional societies, loss of license, or a malpractice suit.
- *Professional experience* – A residency or apprenticeship with an approved organization to gain practical skills.
- *Ongoing education* – Practitioners are required to undertake a minimum amount of self-education on a regular basis, so that they maintain awareness of new developments in their field.

Notice that software development has none of these elements. Anyone, regardless of ability, education or experience can hang out a shingle calling themselves a "software developer," without challenge. Worse, practitioners may behave in any manner they choose, without restraint. The strict ethical requirements of a medical practitioner aim to ensure that the patients needs are best served. In the absence of such requirements, a software developer is free to scheme, manipulate, lie and deceive as suits their purpose –

consequently we see a great deal of exactly this type of behavior in the field.

Integrity

The key concept in any profession is that of integrity. It means, quite literally, "unity or wholeness." A profession maintains its integrity by enforcing standards upon its practitioners, ensuring that those representing the profession offer a minimum standard of competence. Viewed from the perspective of a non-practitioner, the profession therefore offers a consistent promise of a certain standard of work, and creates the public expectation of a certain standard of service.

Individuals, also, are required to act with integrity. It is not acceptable for them to say one thing and do another e.g. to promise to always act in the best interests of a patient or client, but then let personal interests govern their action. What is said and what is done must be consistent.

This cultural focus upon integrity is entirely missing from the field of software development, and demonstrates the vast gap in maturity that exists between our occupation and the true professions. If we are ever to make a profession of software development, to move beyond the currently fractured and uncoordinated group of individuals motivated by self-interest, with little or no concern for the reputation or collective future of their occupation, then some fundamental changes in attitude must occur. We must begin to value both personal and professional integrity and demonstrate a strong and unwavering commitment to it in our daily professional lives.

Think about it – what are your ethical and professional obligations in your current position. Are you fulfilling them? Look to ethical codes such as those offered by the ACM[2] and the IEEE-CS[3], even if you are not a member of these societies. Although not legally binding, they at least demonstrate the sorts of concerns you should be championing in your everyday work. You will find that their central focus is upon always acting with integrity; always representing the best interests of the client. Specifically, you will note that the following behaviors, as commonplace as they are amongst developers, are antithetical to ethical conduct:

- Choosing technologies and solutions because they are "cool", have novelty value or look good on your CV.

- "Going with the flow" or "keeping a low profile: i.e. remaining deliberately distant from or ignorant of issues which affect the quality of service delivered to the customer. You must be willing to voice unpopular facts or express controversial opinions if you have reason to believe that not doing so will compromise the service delivered to a client.
- Distancing yourself from others who are attempting to maintain a minimum standard of work or conduct, so as to avoid any political risk yourself. If you are aware of a challenge to the ethical standards of your profession, you are obliged to defend those standards, even if you have not been directly involved.
- Letting unethical conduct go unchallenged. To observe unethical conduct and say nothing is to offer a tacit endorsement of that behavior. Saying "It's not my problem," "It's none of my business" or "I'm glad that didn't happen to me" is not acceptable. Next time, it may be happening to you.

There's no denying that acting ethically can have a personal cost, perhaps quite a profound one. It would be naive to think that attempts to contradict or combat unethical behavior are not likely to result in some attempt at retribution. Even in professions with legally binding codes of ethics, this is the case. In software development, where it is a moral free-for-all, it is particularly so. Raising ethical objections, voicing unpopular facts, standing up for the client's rights where they conflict with some manager's self-interest – all of these actions bring a very real risk of retribution from offended parties, that may include losing your job. Because ours is not a true profession, there is no protection – legal or otherwise –- for a developer who speaks the truth and in so doing defies authority. Whoever is most adept at bullying, intimidation and political manipulation is likely to hold sway.

I suspect that more than a few of the incidents we have recently seen involving the termination of bloggers for alleged indiscretions on their blogs have been excuses for employers to remove inconvenient employees who threaten the status quo. Although superficially plausible reasons may be offered for such action, they may well be nothing more than an excuse for retribution against the employee for challenges they have made to the employer's unethical behavior.

There Was A Crooked Man

In assessing the personal cost of ethical action, it helps to maintain a broader perspective. In our industry, jobs come and go like the seasons. Due to the prevalence of contract work, many software developers will likely have dozens of employers in their careers. Rather than viewing our work as a series of unrelated engagements, I believe we need to view our efforts as part of a larger process – the maturation of an occupation into a true profession. Seen from this angle, the significance of any particular job (or the loss of it) is lessened and the importance of the over-arching principles becomes more obvious.

As they say, the chain is only as strong as its weakest link. The strength of our reputation and worth as a burgeoning profession is therefore dependant upon the strength of the individual's commitment to maintaining a high personal standard of ethics. The integrity of the whole is contingent upon the integrity of the parts.

Some years ago I read the following statement, which for its truth and boldness has stuck with me ever since:

The best managers are the ones that come into work each day prepared to lose their job.

In other words, unless you remain willing to walk away from a job, the threat of termination can always be used against you, and used as leverage to encourage or excuse unethical behavior. The same reasoning applies to developers as it does to managers. The same ethical obligations and the same obstacles to fulfilling them are present.

In 1985, David Parnas resigned his position as member of a U.S. Defense Department Committee advising on the Strategic Defense Initiative (SDI). He felt, with good reason, that the goals set for the SDI were entirely unachievable, and that the public was being misled about the program's potential. Others urged him to continue, and continued with it themselves, even though they shared his beliefs about the feasibility of the programs fundamental objectives. They reasoned that, even though the desired outcomes wouldn't be achieved, there was good funding to be had that might be put into ostensibly "contributing efforts", and the opportunity was too good to miss. When Parnas resigned, he wrote a series of eight papers [4] outlining both his reasons for doing so, and the fundamental issues about software professionalism that the SDI issue had bought to light. Unfortunately, he have very few men of his quality in our occupation.

Parnas summarized a professional's responsibility in three statements, which I conclude with here:

- I am responsible for my own actions and cannot rely on any external authority to make my decisions for me.
- I cannot ignore ethical and moral issues. I must devote some of my energy to deciding whether the task that have been given is of benefit to society.
- I must make sure that I am solving the real problem, not simply providing short-term satisfaction to my supervisor.

---

[*] First published 7 Aug 2005 at http://www.hacknot.info/hacknot/action/showEntry?eid=77
[1] *After The Gold Rush*, Steve McConnell, Microsoft Press, 1999
[2] http://www.acm.org/constitution/code.html
[3] http://www.ieee.org/portal/pages/about/whatis/code.html
[4] *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley, 2001

From James Dean to J2EE: The Genesis of Cool*

It has always been the purview of the young to define what "cool" means to their generation. In the fifties, cool was epitomized by James Dean. Teenagers rushed to emulate him in looks and manner. Cigarettes, leather jackets, sports cars and a crushing sense of parent-induced angst were the hallmarks by which these youth declared both their distance from the previous generation and unity within their own.

In the sixties, the hippy generation stepped off the path to maturity their parents had planned out for them, put flowers in their hair and went on a drug assisted exploration of their own psyche to the soundtrack of Jimi Hendrix and The Jefferson Airplane. The meaning of cool became a little more diffuse. As an adjective of laid back approval, it still carried the antiauthoritarian flavor of the previous decade; but was broad enough to include almost anything of an unconventional nature.

In the seventies, bigger was better. Wide collars and ties, flared trousers and ostentatious jewelry were the adornments of the young and cool. Disco was king and the Bee Gees were the kings of disco. The definition of cool could only be broadened to accommodate the crass symbols of consumerism that the cultural elite filled their home and their wardrobes with. For the first time, cool was as much about earning capacity as it was about rebellion.

In the eighties, consumerism and technology joined forces to highjack cool from the hands of the kids. It became an adjunct to the management buzzwords and marketing neologisms that littered the corporate lingo. The electronics companies created synthesizers that dominated the music of the decade, and sold them back to the youth who were wondering what had become of cool. "Behold", they said, "this is technology and verily, it is cool."

In the nineties, cool went through its final stage of deconstruction to become the meaningless mouth-noise that we have today. With the unexpected rise in popularity of the Web and its accompanying soap bubble of financial optimism, cool became the adjective of choice for the technically literate. In keeping with their unfettered enthusiasm and cavalier attitude, dot-com entrepreneurs everywhere looked up only briefly from their Palm Pilots to heap uncritical praise upon every new technology and gadget that passed across their expansive desks.

The Future Of Cool

This decade, “cool” means nothing. It is a label applied so ubiquitously and indiscriminately that it could compete with "nice" for the title of “Most Ineffectual Adjective in Common Usage.” The retro punk rockers with their double basses and Gibson Epiphones think they have it. The Feng Shui consultants and the new age drop-outs think it has something to do with Atlantis. The advertising executives and middle managers know that they had it once, but then it slipped between the cushions of their leather lounges along with their ridiculously miniature mobile phones.

But most laughably of all, we the techies think that we have it. Surprised to find that technology is now cool, we feel justified in labeling the geekiest of our enthusiasms with this meaningless endorsement.

Pop quiz: Which of the following are cool?

- Open source
- Linux
- Visual Basic
- Windows XP
- Extreme Programming
- MP3
- Quake
- J2EE
- .NET

There are no correct answers to this quiz, and your response means nothing – unless you voice it with breathless enthusiasm while gazing in a shop window.

In the coming year, cool will lead us everywhere and nowhere, with the following predictable detours:

- Many software projects will be initiated by software developers with a cool hammer looking for some business-case nails to justify their expenditure. Projects thus founded will fail, but not before the developers have had a nice time playing with their new hammers and increasing their market appeal to future employers in search of the latest coolness.

- Many vendors will grunt out another selection of half-baked products that promise a world of coolness but deliver instead a slew of bugs, patches and service packs. The products these same vendors previously marketed as cool will be mysteriously absent from their catalog, although many of the newer products will bare an uncanny resemblance to their predecessors.
- The shelves of technical book stores will overflow with 500 page tomes promising a quick path to mastery of these latest technologies. The speed with which these books are issued and revised will equal or exceed the release rate of the technologies they describe.
- Many legacy systems that have been providing satisfactory service for years will be decommissioned and replaced with systems based on newer and cooler technologies. These replacements will be less reliable than their predecessors.
- Technology selection based on hard-headed empiricism will be viewed as impossibly expensive and time consuming, and abandoned in favor of emotive decision making based on marketing promises and perceived tech appeal. We will be too busy climbing the learning curves of the latest software development gear to have any time remaining in which to quantifying the costs and benefits of doing so. Hamsters … exercise wheels … same old story.

The overall success and failure rates of software projects will remain much as it was last decade, and everyone will bemoan the sad state of software development.

[*] First published 11 Jan 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=43

IEEE Software Endorses Plagiarism*

plagiarize *– take (the work or an idea of someone else) and pass it off as one's own. – New Oxford Dictionary of English*

Ours is an occupation obsessed with invention and novelty. Every week it seems that some new technology or development technique arrives, heralded by a fanfare of hype and a litany of neologisms. So keen are we to exploit the community's enthusiasm for newness that we will even take old ideas and rebadge them, offering them up to our colleagues as if they were original.

Every time I see such reinvention, I feel a certain discomfort. There seems to me something fundamentally wrong with positing work as being entirely your own, when it in fact borrows, duplicates or derives from the work of others.

In science, precedence counts for a great deal and authors are usually generous and fastidious in providing correct attribution and acknowledgement of former discoveries which their own work has benefited from. Indeed, a broad indication of the significance of a paper is the number of subsequent citations that the work receives. In software development, there appears to be rather less respect for the contributions that others make; perhaps even a certain contempt for prior art.

Fail Fast

A particularly egregious example of this disrespect for precedence appeared in the Sept/Oct 2004 issue of IEEE Software, in an article in the *Design* section by Jim Shore called *Fail Fast* [1]. The section editor is Martin Fowler.

Shore describes "*a simple technique that will dramatically reduce the number of bugs in your software*". His technique, which he considers "*nonintuitive*" is to write your code so that it fails "*immediately and visibly.*" This is achieved by putting assertions at the beginning of each method, that check the validity of the values passed to the method's arguments, throwing a run-time exception if invalid values are encountered.

For example, if you write a method for finding the positive square root of a non-negative argument, you make the expectation of "non-negativity" explicit at the beginning of the method, like this:

```
public void squareRoot(float value) {
  if (value < 0.0) {
    throw new SomeException(value);
  }
  // More code goes here
}
```

This technique is the antithesis of *defensive programming*, which would encourage us to make the method as tolerant of unexpected input as possible.

Shore then goes to some lengths to enumerate the strengths of this technique, such as:

- When failure occurs, the result is a stack trace that leads directly to the source of error. Code that doesn't fail-fast can sometimes propagate errors to other portions of the call hierarchy, finally to fail in a location quite distant from the original point of error.
- Reduced use or elimination of a debugger; the messages from the assertion failures are sufficient to localize the error.
- Logging of assertion failures provide excellent debugging information for maintenance programmers who later diagnose a production failure from log files.
- Reduced time and cost of debugging.

There are no citations anywhere within the article; nor does it specify any references. The author (and by extension, the editor) are apparently content to have you believe that this concept is new and original.

Design By Contract

You may well be familiar with the term *Design by Contract* (DBC). The term was coined by Bertrand Meyer, and a full exposition of it may be found in Chapter 11 of his excellent text *Object Oriented Software Construction* [2]. Shore's Fail Fast technique is nothing more than a re-naming of a subset of the concepts within DBC. In short, “Fail Fast” is entirely derivative in nature.

For those who have not previously encountered it, DBC is a technique for specifying the relationship between a class and its clients as a formal agreement [2] – a *contract*. A contract is expressed as an assertion of some boolean conditional statement. When the condition is false, the contract is said to fail; which results in the throwing of a runtime exception.

Broadly speaking there are three types of contracts – preconditions, postconditions and invariants. The *Fail Fast* technique relies only upon preconditions – assertions placed at the beginning of a method that specify the conditions the method assumes to be true. The topic of DBC is fairly involved, particularly with regard to the way that contracts accumulate across inheritance relationships. Meyer's exegesis of DBC is vastly superior to the limited discussion of preconditions (under the new name “Fail Fast”) given by Shore.

Not only does Shore co-opt the work of others, he combines it with bad advice regarding the general use of assertions. Shore claims:

> *When writing a method, avoid writing assertions for problems in the method itself. Tests, particularly test-driven development, are a better way of ensuring the correctness of individual methods.*

This is the purest nonsense. Assertions are an *excellent* way of documenting the assumed state of a method mid-way through its operation, and are helpful to anyone reading or debugging the method body. This was first pointed out by Alan Turing back in 1950:

> *How can one check a large routine in the sense that it's right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.3*

In contrast to Shore, Meyer is generous in his acknowledgement of predecessors and contributors to DBC itself. Section 11.1 of his text has an entire page of "Bibliographical Notes" in which he acknowledges the work of Turing, Floyd, Hoare, Dijkstra, Mills and many others. Indeed, he has delivered an entire presentation on the conceptual history of DBC prior to his own involvement.[4]

Giving Credit Where Credit Is Due

Such misattribution and inattention to precedence as Shore's harms our profession in several ways:

- It is professionally discourteous in that it denies those who develop and originate work their proper credit.
- It discourages modern readers from exploring the history of the concepts they are presented with, thereby denying them an opportunity to deepen their knowledge through exploration of the prior art. Meyer has already expounded the benefits of "fail fast" versus "defensive programming" at length. If Shore's article had appropriate citations, readers would be directed towards this better and more detailed explanation, and would realize that the concept can be taken much, much further through postconditions, invariants, and inheritance of contracts
- It garners false credit for those who ignore the precedence of other's work, encouraging others to do the same – diverting energy into the re-labeling of already known concepts that could otherwise be directed into new areas.
- It creates confusion amongst the readership and obfuscates links with the existing body of knowledge. Central to any epistemological effort is a consistent naming scheme, so that links between new discoveries and existing concepts can be identified. Renaming makes it difficult, particularly for those new to the field, to distinguish new from old concepts.

Conclusion

To have work published in a peer reviewed journal is a significant achievement. It means that one's work has been found to make a worthwhile contribution to the literature, and to be of a high professional standard. By these criteria, the *Fail Fast* article by Jim Shore in the Sept/Oct 2004 issue of IEEE Software should not have been published. The material it presents as being new and original is a superficial (and flawed) restatement of earlier work by Meyer, Hoare and others. It should be cause for concern for us all that a high profile, professional journal should publish work that is derivative and misrepresentative. Those who reviewed Shore's article prior to publication, and the editor/s who approved its publication deserve the harshest admonishment for effectively endorsing plagiarism.

---

[*] First published 2 Oct 2004 at http://www.hacknot.info/hacknot/action/showEntry?eid=67
[1] *Fail Fast*, Jim Shore, IEEE Software, Sept/Oct 2004, pg 21
[2] *Object Oriented Software Construction, 2nd Edition*, Bertrand Meyer, Prentice Hall, 1997
[3] *Checking A Large Routine*, Talk delivered by Alan Turing, Cambridge, 24 June 1950.
[4] *Eiffel's Design by Contract: Predecessors and Original Contributions*, Bertrand Meyer

Early Adopters or Trend Surfers?*

***Q:** What are the most exciting/promising software engineering ideas or techniques on the horizon?*

***A:** I don't think that the most promising ideas are on the horizon. They are already here and have been here for years but are not being used properly.*

– Interview with David L Parnas

Many software developers pride themselves on being up to date with the latest software technologies. They live by the credo "beta is better" and willingly identify themselves as early adopters. The term "early adopter" comes from the seminal work on technology transfer *Diffusion of Innovations* by Everett M. Rogers (1962). He categorizes the users of a new innovation as being innovators, early adopters, early majority, late majority and laggards. Innovators and early adopters constitute about 16% of the user population.

Amongst the software development population, that percentage must be significantly higher, given the technological orientation of most practitioners. Consider the following selection of recent technologies and their respective dates of introduction. Observe how quickly these technologies have become main stream. In about five years a technology can go from unknown to common place. In ten years it is passé?

Technology	**Introduced**
JSP	1998
EJB	1998
.NET	2002
Java	1995
J2EE	1999
SOAP	2000
Microsoft Windows	1993
GUI	1974

Now consider the following software development practices:

Practice	**First Noted**
Source code control	1980
Inspections	1976
Branch coverage testing	1979

Practice	**First Noted**
Software Metrics	1977
Throwaway UI prototyping	1975
Information Hiding	1972
Risk Management	1981

Why is it that after, in some cases, 20 years worth of successful application in the field, often accompanied by repeated empirical verification of their worth, many of these practices are yet to be considered even by the early majority?

Adopting new technologies is easy, but changing work practices is hard. Technologies are "out there" but work practices are distinctly personal. And new technologies promise immediate gratification by way of satisfying the hunger for novelty.

* First published 25 Sep 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=24

Reuse is Dead. Long Live Reuse.*

Reuse is one of the great broken promises of OO. The literature is full of empirical and anecdotal evidence to this effect. The failure to realize any significant benefit from reuse is variously ascribed to technical, organizational and people factors. Observation of the habits and beliefs of my fellow software engineers over many years leads me to believe that it is the latter which poses the principle obstacle to meaningful reuse, and which ultimately renders it unachievable in all but the most trivial of cases.

Hubris is a common trait amongst software developers and brings with it a distrust and disrespect for the work of others. This "not invented here" attitude, as it is commonly known, leads developers to reinvent solutions to problems already solved by others, driven by the conviction that the work of anonymous developers must be of dubious quality and value. Some simply prefer "the devil you know" - figuring that whatever the shortcomings of a solution they may write themselves, their familiarity with it will sufficiently reduce the cost of subsequent maintenance to justify the cost of duplicating the development effort. Evidence of this drive to reinvention is everywhere. Indeed, the collective output of the open source movement is proof of the "I can do better" philosophy in action.

Consider what it is about software development that attracts people to it. In part, it is the satisfaction that comes from solving technical problems. In part, it is attraction to the novelty of new technologies. In part, it is the thrill of creating something that has a life independent of its original author. Reuse denies the developer all of these attributes of job satisfaction. The technical problem is already solved, the new technology has already been mastered (by somebody else), and the act of creation has already occurred. On the whole, the act of reuse is equivalent to surrendering the most satisfying aspects of one's job.

So what degree of reuse can coexist with such a mindset? Certainly we may abandon hope for any broad reuse such as that promised by frameworks. Instead, we may expect frameworks themselves to proliferate like flowers in spring. The greater the scope of the potential reuse, the greater the opportunity to disguise technology lust and hubris as genuine concerns over scalability or applicability.

I believe the only reuse likely to be actually realized is in the form of limited utility libraries and perhaps small GUI components. If the problem the potentially reusable item solves is seen as technically novel or intriguing, then reinvention will result. If there is no entertainment, novelty or career value in reinvention then begrudging reuse may result simply as a way of avoiding "the boring stuff." But as long as developers are willing to use their employer's time and money to satisfy their personal ambitions; and as long as they continue to believe they hold a personal monopoly on reliable implementation, then the cost advantage of reuse will remain a gift that we are too proud to accept.

[*] First published 4 Aug 2003 at http://www.hacknot.info/hacknot/action/showEntry?eid=13

All Aboard the Gravy Train*

"Hype is the plague upon the house of software." – Richard Glass

It is interesting to watch the software development landscape change underfoot. As with many geographies, the tremors and shifts which at first appear random, when more closely examined reveal an underlying order and structure that is more familiar and less mysterious.

Recently, some of the loudest rumblings have been coming from that quarter whose current fascination is the scripting language Ruby, and its database framework Rails. Think back to the last cycle of hype you saw in our industry – perhaps the Extreme Programming craze – and you'll recognize many of the phenomena from that little reality excursion now reoccurring in the context of Rubyism. There are wild and unverifiable claims of improved productivity amidst the breathless ravings of fan boys declaring how cool it all is. There are comparisons against precursor technologies, highlight faults that are apparently obvious in hindsight, but were unimportant while those technologies were in fashion. And above all there is the frenetic scrambling of the "me too" crowd, rushing to see what the fuss is all about, desperately afraid that the bandwagon will pass them by, leaving them stranded in Dullsville, where nothing is cool and unemployment is at a record high.

But this crowd faces a real dilemma, for there are multiple bandwagons just ripe for the jumping upon. Which to choose?

The Web 2.0 juggernaut has been on tour for some time, despite the lack of a cogent definition. The AJAX gang have also been making a lot of noise, mainly because the Javascript weenies can't contain their excitement at being in the popular group again.

But how and why does all this techno-fetishism get started?

Now Departing On Platform One

"Welcome aboard the gravy train, ladies and gentleman. Our next stop is Over-enthusiasm Central. Please be advised that critical thought and a sense of perspective are not permitted in the passenger compartment. Please ensure that your safety belt is unfastened while the red dollar sign is illuminated. We know that you have a choice of bandwagons,

and thank you for your choice to bet the farm upon this one. We promise – this time it'll be different."

The endless cycle of technological and methodological fashions that so characterizes our industry is the result of a symbiotic relationship between two groups – the sellers and the buyers.

The sellers are the parties who are out to create a "buzz," generating a desire for some technology-related product. They include the corporate vendors of new technologies such as Sun and IBM. Alongside them are the pundits and self-promoters who are looking to make a name for themselves. They attach themselves to particular trends in order to cross-sell themselves as consultants, authors and speakers. Hot on their heels are the book publishers and course vendors, who appear with remarkable speed at the first hint of something new, with a selection of 500 page books and offsite training courses to ease your transition to the next big thing.

The buyers are the developers who hear the buzz and are drawn to it. And for many, that draw is very strong indeed, for a variety of reasons. First, many developers are fascinated with anything new simply because it is a novelty. The desire to play with new tech toys is what got many into IT to begin with, and is still their main source of enjoyment in their working lives. For others, the lure of a new technology lies in the belief that it might solve all their development woes (rarely is it stated directly, but that's the tacit promise). It's classic "silver bullet" thinking of the sort Fred Brooks warned against 25 years ago, but which is just as deceptively attractive now as then.

Incoming technologies have the same advantage over their predecessors that opposition political parties have over the governing party; the shortcomings of the existing option have been revealed through experience, but the shortcomings of the incoming option are unknown because nobody has any experience with it. This makes it easy to make the incoming option look good by comparison. You just focus on the problems with the old technology, while saying nothing of the problems that will inevitably accompany the new one. The newer option has an image that is unblemished by the harsh light of experience. The new technology is promoted as a key ingredient of forthcoming software success stories, but those pieces of software are just vaporware, and vaporware doesn't have any bugs or suffer any performance or interoperability problems.

It should also be acknowledged that there is a psychological and emotional appeal to placing such emphasis upon the technological aspect of software development. It alleviates the burden of self-examination and introspection upon work practices. It is much easier and more comfortable to think of all one's problems as being of external origin, leaving one's self blame free. "As long as the problem is "out there" somewhere, rather than "in here", we can just jump from one silver bullet to the next in the hope that maybe this time the vendors have got it right. Heaven forbid that the way we apply those technologies should actually have something to do with the sort of outcome we achieve.

But think of this:

Of all the failed and troubled software development efforts you've been involved in, there is one common element ... you.

Your Regularly Scheduled Program

Some developers enjoy this perpetual onslaught of marketing efforts, for it keeps them well supplied with new toys to play with. But some of us are both tired of the perpetual call to revolution, and concerned for the long term effect it has upon our profession. I belong to the latter group.

The main danger that this ever-changing rush to follow technological fashion has upon us is to distract us from focusing on those aspects of our work that really matter – the people who are doing the work and the working methods they employ. Do you think that the technologies you use really make much difference to the outcomes your achieve? I suggest they are generally quite incidental. To understand why, consider this analogy.

Suppose a group of professional writers gather together for a conference discussing the nature of the writing activity. You would expect them to broach such topics as plot, character development, research methods, editing techniques and so on. But suppose they spent their time discussing the brand of pen that they preferred to write with. If one author claimed "My writing has got so much better since I started using Bic pens" - would you not think that author might be missing something? If another claimed "That book would have been so much better if it'd been written with a Parker pen" - you might again think that the speaker has missed the point. If a third claimed "I write twice as much when I use a Staedtler pen," you might think that the author is probably making things up, or at least trying to rationalize a behavior that is really occurring for emotional or psychological reasons. But isn't this exactly what we developers do when we claim "This project would have been so much better if we'd written it in Ruby" or "I'm twice as productive writing in Java as I am in C++"? In other words, our focus is all wrong. We're preoccupied with the tools we use, but we should be focused on the skills and techniques with which we wield those tools.

At the organizational level, this fixation with novelty often works to create a bad impression of IT's capabilities and proclivities. If those that make the strategic technology decisions for a company are the type to get carried away with the latest fads, then that company can find itself buffeted by the ever-changing fashions of the technical industry, always switching from one "next big thing" to another, with no concern for long term maintenance burden and skills investment. It is easy to create a portfolio of projects implemented in a broad range of diverse technologies, requiring an equally diverse set of skills from anyone hoping to later maintain the project. A broad skill base is seldom very deep, so staff become neophytes in an ever-increasing set of technologies, none of which have been used for a sufficient time for them to gain a high level of expertise. From an outsider's perspective, the IT section seems to be a bunch of boys playing with toys, terminally indecisive, that for some reason needs to keep re-implementing the same old applications in progressively newer and cooler technologies, though successive reimplementations don't seem to be getting any better or more reliable. It seems that every six to twelve months they suddenly "realize" that the technologies they're currently using aren't adequate and a new technology direction is spawned. All that is really happening is that the novelty of one technology selection has worn off and the hype surrounding some new novelty is beckoning.

Think of the organizational detritus this leaves behind. You've got legacy VB applications that can only be maintained by the VB guys, legacy J2EE systems that can only be maintained by the J2EE guys, a few .NET applications that only the .NET guys can comprehend, and that Python script that turned out to be unexpectedly useful, which no one has been game to touch since the Python enthusiast that wrote it resigned last year.

How many companies, do you suppose, are now left with monolithic J2EE systems containing entity beans galore, that were written as the result of some consultant's fascination with application servers, and their compulsion to develop a distributed system even if one wasn't required. And how impressed are the executives in those companies who find themselves with an enormous, sluggish system that appears to have gone "legacy" about five minutes after the consultants left the building. Can we be surprised at their cynicism when they're told their system will have to be rewritten because it was done poorly by people who didn't really understand the technologies they were working with (how could they –

they were learning as they went). How can they leverage their technology and skill investments when both seem to become irrelevant so rapidly?

What's The Better Way?

Thankfully, it doesn't have to be like this. But avoiding the harmful effects of technology obsession requires some clarity.

At the organizational level, it requires senior technicians to have the maturity and professional responsibility to put the interests of the business before their personal preferences. It means developing technology strategies and standards based solely upon benefit to the business. It means remembering that there is no ROI on "cool."

At the individual level, it means adopting a skeptical attitude towards the hype generated by vendors and pundits; and turning one's focus to the principles and techniques of software development, which transcend any technology fashion. Your time and energy is better invested in improving your abilities and skills than in adding another notch to your technology belt.

[*] First published 27 Aug 2006 at http://www.hacknot.info/hacknot/action/showEntry?eid=89