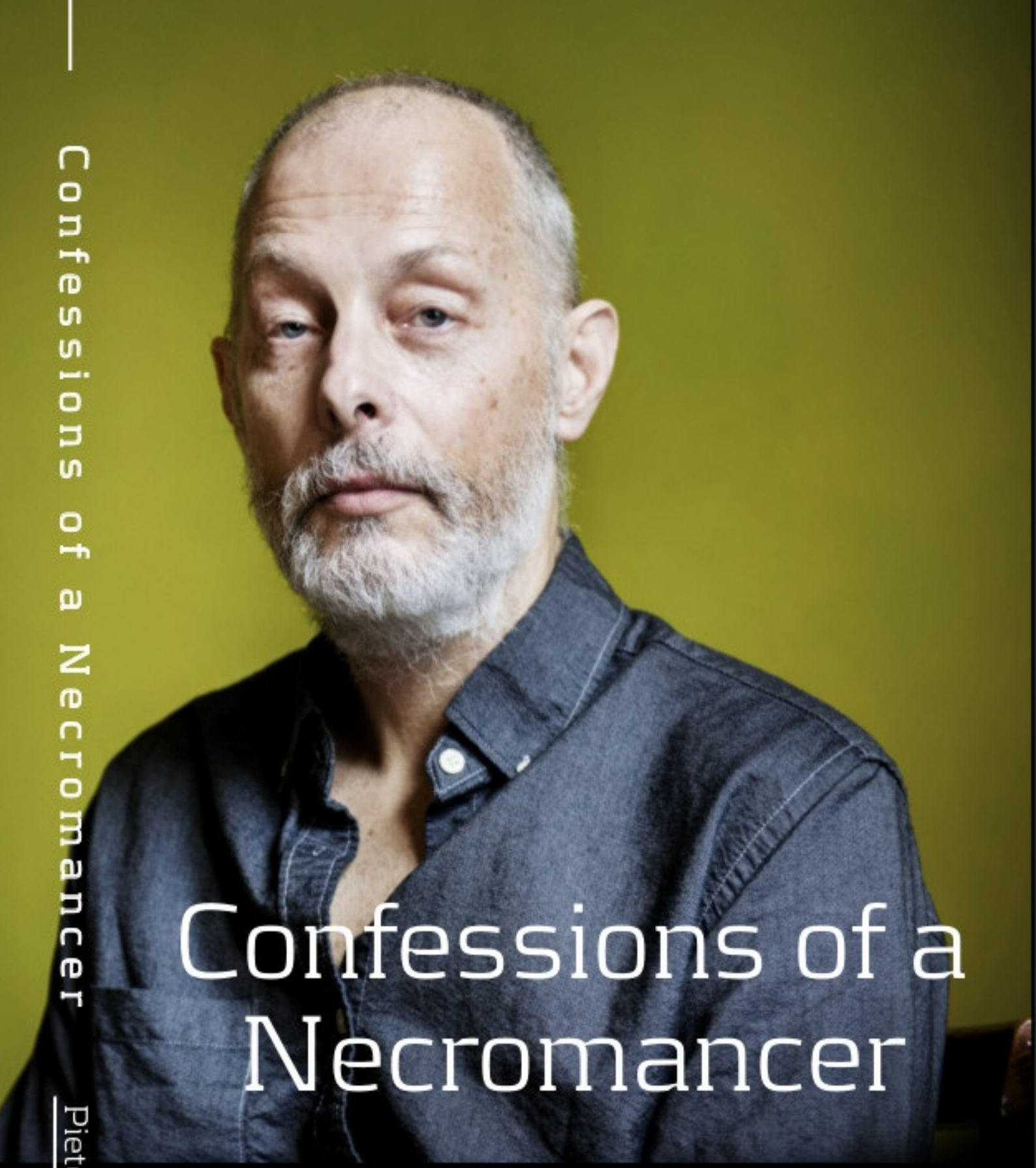

Confessions of a Necromancer

And Other Stories

by

Pieter Hintjens

Confessions of a Necromancer
Pieter Hintjens

Table of Contents

Confessions of a Necromancer - nd other stories

Save to Kindle

Source repo is https://github.com/hintjens/confessions, pull requests will be ignored.

Cover font: Kontrapunkt by Bo Linnemann, Kontrapunkt A/S. Text fonts: EB Garamond by Georg Duffner, MonospaceTypewriter by Manfred Klein..

Pieter Hintjens is a programmer, writer, and thinker who has founded many on-line communities. In 2005 he led the European fight against software patents, helping thousands of activists to organize on-line. In 2007 he founded the successful and broad-ranging ZeroMQ community.

Other books by the same author: "ZeroMQ - Messaging for Many Applications" (O'Reilly), "Culture and Empire: Digital Revolution" (Amazon.com), "The Psychopath Code" (Amazon.com).

This is a collection of stories and articles from my blog. Some are long, some are shorter. Read them in any order, though best from left to right along each line and top to bottom on the page.

Why make a paper edition of digital writing? Perhaps for the simplest of reasons. When you make a gift to someone of a book, wrapped in nice paper, that feels intimate and tangible. Writing "http://hintjens.com" on a piece of paper and giving that does not seem equivalent. And we like to give gifts, most of us.

Anyhow, I wrote these stories at different times. Some are an attempt to speak truth, and some are just persiflage. Most are from late 2015 until my death. That is coming soon. On 12 April my oncologist told me my old bild duct cancer had returns. Six months later, I'm starting the formalities for euthanasia (they are not large) here in hospital.

It has been hard to put a finger on death, which keeps dancing out of range. So this book isn't as polished as it might be. I've broken it into several sections, creatively called "chapters". In the last one I've collected the various articles I wrote during those six months directly related to dying.

This book should not make you sad. It aspires to be a celebration of life.

My first act upon getting confirmation of my impending doom was to organize a party/wake/riot/bbq at my place. This took place on 4th/5th June, over a long warm weekend. It was one of the best parties I've ever been too, in my own mind. I can't describe it. If you were not there, you missed something. Lesson learned: if you get a terminal diagnosis, hold a living wake! It is way, way better than a funeral for giving people a chance to say goodbye.

And with that, onwards to my first pieces, the slightly unfinished "Confessions of a Necromancer."

Chapter 1. Confessions of a Necromancer

Warning: Indulgences Ahead

If you wanted a brief, snappy piece on the meaning of life, skip this one. I'm writing for my own pleasure this time. This is less of a blog post, more of a novella. And it's all about me, the smartest guy in the room, always with an answer, almost impossible to beat in an argument.

This isn't an autobiography, I'm not going to talk about my family life, or our idyllic childhood years spent in Dar es Salaam. I'm not going to explain why my first languages were Lingala and Swahili. I'll spare you my school years, spent in the cold, dark embrace of the Scottish highlands.

Instead, this is a professional road trip, for my fellow programmers. OK, not entirely.

My first software products ran on a computer with 5,120 bytes of RAM, 1,536 of which were dedicated to storing the contents of the screen. If someone tells you that you can do a lot in 3.5KB of RAM, they're lying. You can't do shit with that. However my next computer came with 64KB of RAM, at the same price. You try filling that up, one assembler instruction at a time. And that's when I realized, these things aren't toys any more. They can do real work. They can make real money.

I've worked on a lot of different systems. More, I've worked in the weirdest possible projects, with crews of all sorts. There has always been a thread of insanity in our business. We're hooked on doing the impossible. Clients lie to themselves that they've hired a team that know what they're doing. The team lie to their clients that they're in control. The sales people lie. The marketing department lies.

Most of the time, most large software projects fail, and this is still true in 2016. Yet for most of my career, my special talent was to make projects work, no matter how impossible the technical challenges. I am a really good technical architect, able to understand systems at the lowest and the highest levels. I can write all the code myself, or I can explain to people exactly what to make, and it all fits together like laser-cut blocks.

As I wrote much later in "Social Architecture,"

> I've come to think that the very notion of individual intelligence is a dangerously simplified myth.

It took me decades to realize that technology is a slave to personality. It doesn't matter how good the design, when there are unresolved problems in the organization.

And so gradually I shifted from technical architect to social architect. From caring about technical designs to caring about people and the psychology that drives them. Because in the end, this is what seems to make the difference between a working project and a failure.

The Microcomputer Era

I didn't plan on becoming a programmer. Age six or seven I wanted to be a writer. I had Roget's Thesaurus, and read it like a novel. Words, each a story, intertwined in endless trails. Eight years of boarding school beat the dreams out of me, and at 17 I dutifully collected good exam results and started thinking, what should I study at university?

My career advisor suggested computers as a lucrative option. Shrug. Seems fine. Cambridge offered a computer science course, yet my exam scores weren't good enough for that. Second best rated was York, which had a computer science + mathematics mix. I applied, went for an interview, and was accepted.

English universities offer a 3-year BSc course. I was a lousy student. The maths numbed my brain. Endless theory about encoding, Turing machines, database normalization. And then finally, a chance to write some code. Work out the PDP-11 assembler code on paper. Enter it by hand using console switches. Run it, and get boxes to move around on the screen. Yay!

About this time the Sinclair ZX80 came out. It was a ridiculous thing, tiny and light, with a horrid rubber keyboard. Yet it was real. I'd go with my friends to the store, and write small BASIC programs on the spot. Here, a game that let you chase a star around the screen with the arrow keys. People would watch me write these, and then ask if they could play.

For two years I struggled through subjects that grabbed my attention like a slug grabbing a basket ball. My tutor was patient and miserable with me. My failing grades just piled up, academic debt. There was no repeating a year. I could change to a different degree if I wanted to. Instead, life gave me a better option.

In the spring of 1981, the VIC 20 hit the shelves in the UK. I asked my mother if we could have one. We were not wealthy, and the computer was more expensive than the Sinclair. Yet it was a different beast, with color and sound and potential. She believed in me, as she has always done, and somehow found the money for the computer.

At first I copied games from magazines, typing them in -- as we did, in those days -- and then playing them. With a VIC cassette recorder, it became possible to save and reload them. Then I started improving the games, adding graphics and sounds. The VIC chip was

powerful, if you were able to figure it out. Then I wrote my own games, mixes of assembler and BASIC. And then I started selling these.

There is a kind of shock, the first time you make something with your own hands, and exchange it for money. I bought empty cassettes, drew and copied inlays, and cranked out my games. A quarter page ad in one of the main magazines cost 125 pounds for three months. One ad impression would produce maybe fifty sales, at five pounds each. When I had a new cassette I'd send a mail flyer to existing clients.

By the summer of '81 I had sold a few hundred games, and attracted the attention of Commodore, who sent me a shiny new C64 free of charge. Computer, disk drive, monitor, and printer! I called my tutor and said, look, Bill, I'd really like to take a year off university to write and sell video games. That OK with you? Sure, he said, and did the necessary.

And so it was that I started my first business. The C64 had vast capabilities, poorly documented. I worked through the reference manual until I knew it by heart. There was an assembler cartridge yet it was clumsy, so I wrote my own assembler tool. And then I started writing a new game. Just a standard shoot 'em up, yet crazy fast and fun, and I was coding day and night to get it finished.

At the time, there was nothing in the market for the C64 except the old nasty BASIC games hastily ported to the larger screen. So I was working like a crazy to get a large-scale production line working. I'd designed and printed full color inlays, fifty thousand pieces.

Putting the cart firmly before the horse, I'd even written a natty copy protection scheme that I was so proud of. Your standard cassette held a BASIC program that you would LOAD, and then RUN. You could as well SAVE it again, to a blank cassette, and people did this a lot. It meant that a popular game get widely shared among computer clubs and high schools. God forbid!

The standard LOAD/RUN thing also seemed clumsy, and I wanted to make it smoother for the player. I'd found that you could format the saved data so that it loaded into a specific address in memory. I crafted a block of data that loaded itself into low memory, and overwrote the stack. When the LOAD routine finished, it popped the address off the stack and hence jumped into my code, which was a special loader. That pulled the rest of the game off cassette, loading it into memory, and then running it.

Many years later my friend Jonathan told me other C64 games used the same scheme, and he copied the games using a special cartridge that simply took a snapshot of memory when the game was loaded and ready to run. Pirates 1, Hintjens 0.

I was working with an audio cassette duplicator in Port Seton, and we'd cracked how to produce software cassettes at high speed. We could produce several thousand a week per machine. It was all ready to go, and then I made a fatal mistake that taught me my first big

lesson about the corporate world.

Instead of running the ads and shipping out my software, I showed my new game to Commodore.

They loved it and told me they wanted to distribute it. At least UK wide, and possibly all of Europe. I was spell bound. My own sales would be in the tens of thousands. Commodore would be able to sell ten, a hundred times more.

So I told them "OK" and paused my own sales. Boxes of cassettes sat in my room. I did not run my ads. And Commodore told me they were working on it. And so I waited.

As I waited, other games started to appear. The large video game companies had finally gotten their act together, and were attacking the C64 market like a pack of killer whales attacking a young baleen whale. Full color ads promised everything. It became pretty obvious that my window had closed.

Commodore, I asked, where art thou? What the heck is going on? Finally they came back to me. "We're not going to distribute your game after all, thanks," they told me. Incompetent or malicious, who knows. I cursed them, and shut down my business.

As it turned out, it was also summer, and so I went back to university to finish my degree. At least I had a personal computer system to work on, and lots of experience in 6502 assembler. The 6502 chip was simple and fast, with a minimal instruction set. In some ways, a precursor to the RISC CPUs that dominate today's world.

My tutor Bill gave me Loeliger's "Threaded Interpretive Languages: Their Design and Implementation," and I decided to make this my thesis topic. The result was a lovely Forth-like language sat atop that brutalist 6502 assembly language. Fast enough for video games, and such fun to program! Every programmer should make a Forth at some point.

Sitting in my room coding day and night felt right. Whereas I'd spent my first two years avoiding hard work, now my brain was addicted to it. Vaguely, I realized there were courses I should be attending. Mostly, I ignored them, and they ignored me right back. My code got too large for the 170KB floppy drive so I stripped off the comments and carried on coding.

When it came to final exams, I sat with my friend Nigel and we skimmed through the course material, making notes. A few hours was enough to read a years' material, summarize it, and digest the summary. By luck our exams were always in the afternoon. It was a blur. One exam a day for two weeks. I came first or second in every one. The thesis committee asked me why I'd not commented my code and I shrugged. Did it matter? Look, let me demo you a couple of games, and walk you through the core threading logic.

The two years of fails brought down my overall degree yet in my entire career not a single person ever, once, asked me what I'd achieved at university. It was just never relevant. There's a big lesson here. Many of the top students in my course went off to Silicon Valley. I slid under the radar and moved, despite my best intentions, to Belgium.

Working for the Man

The Man got me straight out of university, for in those days we still had mandatory military service in Belgium. I'd lived so long in the UK, and was naturalized (I still have a UK passport), yet Belgium demanded its cup of blood, and it got me.

Military service was actually fun. In boot camp we were split into the intellectuals, and the lifers. People like Henri, a nuclear physicist, and myself wore the glasses and obviously had the nerd soft hands. The lifers were young professional soldiers, 16 or 17, who cleaned and scrubbed and marched and drilled. We nerds sat around the barracks, happily useless. They were not going to give us guns.

Ironically, I was already an excellent rifle and pistol shot, with prizes from Bisely, where my school shooting team had competed every year. I didn't tell them that, mainly because I spoke no French and no Dutch.

I worked at the national map making institute. They had a large project to make a digital map of all of Belgium, for the US air force. They carefully scanned in paper maps, tracing the railways, roads, city outlines, canals. The goal was to make maps for cruise missiles.

My boss showed me the team, their GCOS minicomputer, the IBM terminal running on a time-sharing system somewhere, and told me, "your job is to help us make the tapes."

Turns out the USAF wanted the map data on magnetic tapes. We could send data from the GCOS machine to the IBM, though I completely forget how that worked. Working in PL/I on the IBM, I could load in the map data and write a tape.

Ah, but there was a catch. I was not the first person to try to make tapes. The USAF systematically returned the tapes as "unreadable." After several years of mapping, the project was jammed on this one point. So being the smart young thing I was, I investigated. It turned out the USAF were using a UNIVAC system, which unlike the IBMs, the GCOS, and every normal computer in existence, used seven bits per byte instead of eight.

Magnetic tapes were coded with 8 bits in a stripe, perpendicular to the tape direction. These stripes were written like this "||||||" along the tape. It struck me that the UNIVAC was probably just reading seven bits at a time, instead of eight. So the first stripe had one byte, plus one bit of the next (seven-bit) byte. So I staggered the bits like that, and we sent off a tape to the USAF.

It came back after the usual three or four weeks, with a new error message: "Bad data format." Bingo. Now I went back to the documentation and figured out what the real mapping data should look like. What we were sending from the GCOS wasn't even close.

Some weeks later (I got sidetracked into writing a small compiler on the GCOS that could access more exotic system functions and give me unlimited priority to run my heavy conversion programs) we had a new tape. We sent it to the USAF, and a few weeks later came the reply, "Valid."

At which point, all hell broke lose.

What I didn't realize was that the project had some years of budget left. The mapping was mostly done, and the team mostly idle, and happy with it. Lacking any way to send valid tapes, the Belgians simply kept pushing back the date, and collecting their sweet, sweet USAF cash. (I am speculating that it was sweet, because I was being paid a conscript's wage of 45 BEF or about 1.10 EUR per day.)

My boss told me I could take the rest of the week off. Actually, don't come in every day, or unless you feel like it. I shrugged and started my new regime of two half days per week.

There is a lesson here too. You need to follow the money, and understand how it flows. I could have walked away with my own personal GCOS system, if I'd played it smart.

The second lesson, which many a foreigner has learned in Belgium, is that it's an easy country to come to, and a hard one to leave. For reasons, I decided to stay in the place and find a job.

Still Working for the Man

I found a job at Sobemap, which was in 1984 the largest software consultancy in Belgium. I was hired by a large good natured man called Leif, who was looking for a fellow spirit to help him write software development tools. This was in the days of COBOL and weird systems. Sobemap had a commercially successful accounting system, EASY, which they maintained on seven or eight different platforms, with a team for each platform. Leif's job was to build the basis for a single portable EASY that could run on everything.

Turned out, though Leif only told me this a few months ago, that the main objection to hiring me was my age. Too young. I guess it was a high risk project and they hoped for someone with relevant experience. Needless to say, the number of people building cross-platform COBOL frameworks, worldwide, was approximately zero. We were the first to even think of such a mad thing.

It took only a few months to build the basic layers, and then my job was to make those work on new, exotic systems. I'm talking Siemens BS/2000, IBM S/36, DG AOS, MS-DOS (once we discovered the wonderful Realia COBOL compiler for the PC), and IBM MVS/CICS. The core routines had to be in assembler, to be fast enough. That was when I discovered C, on the DG and the VAX and then on the PC.

All these systems died, one after the other, over time. Good riddance. The inconsistency was incredible. Every machine had its own concept of file structures, organization models, compilation procedures, and so on. Imagine learning ten different varieties of Linux, each from a different planet.

No matter, we learned to dive under the consumer level OS and language, and into system internals. That was inevitably the only way to get the performance and behavior we needed. On a BULL TDS (short for tedious) system I wrote a memory manager, in COBOL no less, to fit large programs into the pathetic excuse for "main memory" that system provided. On IBM S/36, the same, in assembler, to swap megabytes of COBOL code and data in and out of the 64KB main memory.

Leif and me, helped by various people who came and left, used our portability tools to build a powerful set of tools: editors, code generators, reporting tools, and so on. Developers loved these tools. You could develop and test on a PC, at home, and the same code would run unchanged on an IBM mainframe. We had cracked the problem of "write once, run anywhere," some years before it became a fashionable problem.

The IT managers of our clients didn't share our passion for portable code. As one IT manager told us, "I've just spent so much on a new VMS system. Why do I want portability?" Logic doesn't apply, when people work on pseudo-mystical belief. A very few IT managers saw the potential, used it, and built their empires on it. Yet mostly we had few external clients and direct sales.

All this time, the company kept changing and shifting. The smart, nice guy who hired Leif and then me left for a startup. We were sat on by a succession of incompetent, arrogant illiterates. Despite delivering success after success, there were no pay rises, no bonuses. Eventually after five years, our team called it quits and we left the firm, each to go separate ways.

Sticking it to the Man

One taste of employeeness was enough for me. I did some research, found an accountant, and became an independent consultant. My first gig was for a trade union organization. They did not pay much yet they treated us with respect. Jonathan Schultz and myself built a project management system, using that awesome COBOL framework.

Somewhat adrift, I worked on various projects with old contacts. We built monitoring software for PCs for a UN transport project in Africa. Set-top clients and servers for a cable TV company. Software to produce and decode shipping manifests, using a prehistoric format called EDIFACT. Mostly I worked in C, using Turbo C on MS-DOS. And a lot of x86 assembler, using MASM, the Microsoft assembler.

Then I got a call from my friend Hugo at Sobemap, now renamed to Sema. "We need your help to make this project work, what's your daily rate?" I quoted a comfortable figure. They agreed. And so I went back to work for the same projects, earning five times more.

I learned one important lesson: the more your clients pay you, the more they appreciate you and listen to what you say. The same CEO who'd walked past me in the corridor as if I was a tattered old poster now stopped and shook my hand. "Glad to have you back with us, Hintjens!" he said.

And so I found myself back in the world of enterprise software projects, providing that special skill we never really had a name for in our industry. Plumbing. Infrastructure. Wet work. Black magic. Making it possible for mediocre developers to build really large applications that worked well even under stress.

It was 1991, and my first major project was to port our tools to VAX ACMS. The project was signed and sold before I had a word to say. The IT manager was one of those who knew Leif and me, understood our blood-and-guts approach, and loved it. He had bet his career and arguably his firm's future, on us succeeding with an impossible project.

So here was the challenge. The client wants to build a travel reservation system to serve tens of thousands of agents in offices across Europe. These agents would log into the system, query availability, make bookings, and so on. The backend for this system would be a cluster of modern, cheap computers called a VAX. These cost a fraction of the IBM mainframes that the industry still relies on. It has cheaper memory and disks than IBM's dinosaurs (though not as cheap as PC hardware). It runs a modern operating system (VMS) that is fast and flexible (though not as nice as UNIX). DEC's wide-area networking was decades ahead of IBM (though not as nice as TCP/IP).

The only problem is that IBM's mainframes have long ago cracked the problem of sharing one mainframe between tens of thousands of users. Whereas DEC, the firm that makes the VAX, has not.

If you want to get technical, IBM used smart 3270 terminals that dealt with all keystrokes locally, and interrupted the computer only when there was a screen of data ready. Somewhat like a clunky web browser. The mainframes then run a "transaction processing system" called CICS that pumps these screens of data through applications, with all state kept out of the process. One application can thus serve masses of users.

The VAX on the other hand used dumb VT terminals. These sent every single keystroke back to the computer to deal with. Each terminal, and its user, have their own process. It's the same model that UNIX and Linux use. Five terminals, five interactive "sessions", five shell processes. So VAX gave a much nicer flow, and you could do things like scroll smoothly (impossibly on a 3270). On the down side it could not deal with anything like the number of users at once. Each of those processes eats up virtual memory.

DEC's proposed solution was to use smaller front-end MicroVAXes, each capable of handling around 50 users. These would then talk via some magic to the back-end cluster. The client did some maths. Two hundred front-ends and ten thousand interactive licenses. That cost several times more than the rest of the project together. They might as well buy an IBM mainframe.

Which was when someone decided to bluff the project, sign it for the budget the client was willing to pay, force DEC to accept whatever solution we came up with, and then hire me to make it all work.

Since no-one had told me it was impossible, I took the documentation and a test system, and began to play with DEC's transaction processing framework, called ACMS. This was the closest DEC had to an IBM CICS-style transaction processor. Think of a container for server applications. You could start and stop apps, and then send them events from other processes, even across a network.

So far so good. Then I looked at VMS's asynchronous I/O system. To get consistent and fast response in a real time system, you have to work asynchronously. You can never wait on something happening.

I'm going to explain this in a little detail, because there are lessons here that still apply today. If you want to make truly massive systems, it's worth understanding the transaction processing (TP) model. Today I'd explain it as follows:

- Your application consists of a collection (hundreds, or thousands) of services. These are connected either directly or indirectly to your front-end UI.
- A service runs with N instances, to provide scaling for many users. The TP starts and stops instances as needed.
- The services typically work with further back-ends, databases, and so on. This is not the concern of the TP system.
- A single service instance works with a single "transaction" at once. Be careful: the term "transaction" is over-loaded. Here, it means a request to do some work, where the work comes from the UI either directly or via some other layers. It is often used to describe a package of work done by a database. The meaning is similar, yet not the same thing.

- Services hold no state for UI sessions. If the service transaction needs state, it is held by the TP, and passed to and from the service, with each call.

This is actually close to the "actor model" that we aspire to with asynchronous messaging.

OK, back to asynchronous I/O. Let's say I'm waiting for the user to press a key on their VT220 terminal. In synchronous code I'd make a system call that waits for input. At that point my process is swapped out, and effectively dead until the user presses a key and input arrives.

An asynchronous call (called an "AST" on VAX/VMS) is a bit different. It does a similar "wait for input" call, and adds extra arguments. It says, "when you are ready, call this function, and pass this state." The critical difference is that after the call, the process isn't suspended. It continues right along with the next instruction.

So you could for instance wait for input on 100 different terminals by making 100 AST calls in a row. You could chain to a single function, and pass the terminal ID as state. Then your process could explicitly suspend itself. Each time a user pressed a key, the process would wake up, the function would get the event, and process it.

This event-driven I/O model is still how we build really fast multi-threaded servers today. However a server typically only deals with one kind of event, namely network traffic. Perhaps three: input, output, and errors. To write an ACMS front-end server, you need to deal with rather more I/O events. At least:

- System calls to resolve logical names, terminal IDs, network names, and so on.
- System calls to read and write to disk.
- Events coming from terminals.
- Events coming from ACMS, typically "finished doing this work."

So you don't have a single event handler, you have dozens or hundreds. This makes the server design horrendous. Imagine writing a large program consisting of tiny functions, where each specifies the name of the next function to call, as an argument.

Being a clever wunderkind, I figured out a solution. One of our COBOL tools was a state machine designer. A neat thing: you describe your program flow as a state machine, and it turns that into code. I'd started rewriting that in C, a project that ended up as Libero.

Using Libero, I could write the flow of execution as a state machine, and generate C code to make it run. In the core of this C code is a generic AST handler that can deal with all possible events. It knows the state machine and can just call the next function itself.

So here is a piece of the state machine, which initializes a new terminal session. What this means isn't important. The point is you can read it, the words kind of make sense, and you're not writing crunky chained AST code:

```
Have-Lat-Connect:
    (--) Ok                                 -> After-Sign-In
          + Spawn-New-Lat-Thread*
          + Move-Thread-To-Active-Pool*
          + Get-Terminal-Port-Name
          + Get-Terminal-Characteristics
          + Set-Terminal-Characteristics
          + Write-Greeting-Message
          + Assert-Iosb-Okay
          + Init-Connection-Parameters
          + Acms-Sign-In
          + Check-Acms-Status-Block
    (--) Error                              ->
          + Signal-Io-Error
          + Deassign-Terminal-Channel
          + Terminate-The-Thread
```

So I presented this approach, with some examples and test results, to the client. They asked a respected consultancy firm (Andersen) to check it. The two consultants who did that work had previously worked for DEC and were experts in ACMS server design. I think they'd helped write the standard ACMS front-ends. After some days of explanations, they went off to write their report, which came down to "the proposed approach is insane and will not work."

Somewhat later we discovered that Andersen had been hoping to get the contract themselves. I learned an important lesson about consultants: they are professional liars.

The IT manager of the client took the report, threw it in the trash, and told his management, either we go with this option or I quit and you can forget this project. Since this man had built the original system on IBM CICS, and brought most of his team into play, that was a serious threat. Management caved, Andersen were kicked out, and we got the green light.

The system we built worked as designed and went live on time in 1992. It grew to handle multiple tour operators, with front-end servers happily dealing with two thousand terminals each. We maintained and extended the system every year for a decade, little by little. When I stopped working with Sema, the client paid maintenance to me directly, until the system was finally decommissioned in 2010.

The lesson here is, if you have the trust of your client, and s/he has real power, you have done half the work already.

This was a perfect software project. Working with good matériel; decent hardware and a non-insane operating system. Working with a smart team that know their stuff, and a client who likes and trusts you. With total freedom to build things the right way. Where the hardware and software bends to your will. Where you can take the risks, analyze them, and design them away.

Little did I know how rare such projects are. Shortly after the tour operating system went live I found myself in an insurance company in Brussels, looking at an ancient terminal the size and shape of a large microwave oven. Carved in the bezel was the ominous word: "UNIVAC."

Shit Goes Downhill

I imagine the conversation went thus: "You're saying you can make the EASY accounting system work on our existing UNIVAC mainframe?" followed by a confident reply, "Of course! It's portable! Now if you'll just sign our framework agreement on licenses and rates," followed by a dryly muffled laugh and the scratchings of quill on parchment.

Whereas the VAX project was a smart bet on new technology, the insurance company was in the midst of a fight between old and new, and we found ourselves on the wrong side of history. The old guard was defending their UNIVAC mainframe (and its not trivial budgets) at all costs. The new guard were trying to push the IT infrastructure towards UNIX.

The first time I saw our client -- the incumbent IT manager -- he was slumped in his chair, melted from fatigue and age, a cigarette in his hand, ash and stubs on his desk. Here is a man in the terminal stages of a truly horrific disease, I felt.

I'm sure that there was a time when UNIVAC made relatively excellent computers. I'm sure someone, in the early 1960's thought, "seven bits is going to be enough for anyone!" I'm sure at some point the slogan "America's First Commercial Computer System!" was spellbinding for customers. But not Brussels in 1992, please no.

By 1992 we'd invented the Internet and were starting to see the first Linux distributions for PC. I had an Yggdrasil distro on CD-ROM that included, wonder of wonders, a free C compiler. Linux was not just a real OS, as compared to the kindergarten MS-DOS, with proper virtual memory and tools. It had the same shells (ksh, bsh) as the Big Iron UNIX boxes we were starting to see in companies.

And here we were, with a system pulled through time like an Automatic Horseless Carriage from 1895 trying to compete in the 1992 Grand Prix.

Leif and I briefly looked through the system docs, and played with the terminals. I looked at the microwave-with-keys thingy and said that it seemed rather ancient. "Oh, it gets worse," said Leif, looking at a page in the manual. This was going to be a familiar phrase. "Oh, it gets worse!"

I'm not going to bore you with detail. Just one small piece to show how bad it was. The UNIVAC terminals were like the classic IBM 3270s, block mode devices that sent the user input as a single block of data, back to the mainframe transaction processor. Something like

a HTML web form, except sending field/value pairs, it would send row/column/value tuples. Fair and good.

The first problem we hit was that the large Enter key did not send anything. Oh no, that would have been too obvious. Instead there was a separate SEND key. OK, we guessed UNIVAC users were used to that. More fun though, the terminal only sent up to the cursor, no further. So if you typed some data, and then backed up to fix an error, and then pressed SEND, you lost half your input.

The terminal had function keys that we could program by sending commands together with screen data. So our hack was to add an invisible input field on row 24, column 80. Then, F1 would move the cursor to that field, do a SEND, add a code for "F1" (our apps used sixteen function keys for navigation), and then move the cursor back to where it had been.

The worst part, which is hard to wash off even after years, is that we were actually proud of this. We'd made it work. There were dozens of other "WTF?" moments, yet eventually EASY could run on this prehistoric system. Leif and I made our excuses and found other projects to work on.

Taking Money off the Table

Around 1992 Sema was building a reservation system for a high-tech fun park in France called Futuroscope. Part of the challenge was to send bookings to the various hotels that clustered around the park itself, and sat further afield.

Before we automated it, bookings were faxed by hand. This was before the days of office laser printers. Each night (or perhaps twice a day), a batch job ran that printed out all hotel bookings on a "line printer." This produced a large "listing," perforated so it could be torn into individual sheets.

Some poor fax jockey took got listing from the Computer People, ripped off the header and footer pages, separated the rest into individual pages, and chopped these using a guillotine into A4 width, so they would fit into a fax machine. Or perhaps they faxed them sideways, breaking all the rules of fax decorum of the time.

I can't imagine the joy of entering each fax number manually, waiting for the modems to connect, feeding in the sheets, and sorting the bookings into "done" and "failed, try again in 30 minutes," and "hotel fax seems borked, call them to see what's wrong" piles. Especially in peak season, when the booking system would spew out hundreds of bookings a day, and even a single misplaced booking would result in drama. If you've never seen a Parisian family of five arrive at a hotel where their booking didn't get through, they make the citizens of San Jose look polite and charming by comparison.

Free software fixed this and destroyed the "fax jockey" position, at least in the Futuroscope.

Futuroscope were looking at commercial bulk fax systems. These were extraordinarily expensive, thousands of dollars just to get started, and more depending on capacity. We found this distasteful especially since by this time a fax modem (a small external box that plugged into a serial port) was maybe a hundred dollars. On Windows we were used to software like WinFax that could send faxes using a special printer driver.

Also we did not want to have to interface with these beasts, which used bizarre proprietary software that I knew was going to cause us needless pain. And like I said, if the faxes didn't work, the whole system was suspect.

So I looked around and found an free software package for UNIX called Hylafax. Together with another free software project called GhostScript, we could create nice (i.e. using proper fonts, and with the Futuroscope logo) bookings, and send them to the hotels entirely automatically.

My idea of using free software was received with solid, head-shaking skepticism. This just wasn't how things were done. What sold my proposal were two arguments. First, that every franc the client spent on expensive fax machines was a franc we couldn't invoice. Second, the fax subsystem was so important that whomever supplied it would become an important vendor. Surely that should be us (Sema) and not some random box seller.

OK, Sema said, if you can make it work, we'll sell it to Futuroscope. So I downloaded the source zips for the packages, built them, and tried them out. It was all surprisingly simple. The hardest part was finding a fax modem that would work with the UNIX server, so I started work on my PC using its fax modem card. This was in the days of dial-up and I paid around $2,000 a year (in today's money) to one of Belgium's first ISPs, in Leuven, for Internet access.

It turned out that a single fax modem was able to handle peak traffic, especially since there was no time lost messing about entering fax numbers and feeding paper. Hotels never replied by fax, so there was no need to handle incoming faxes. All we needed was queuing of outgoing faxes and a way to know if they failed.

Which it turned out, Hylafax mostly did for us. It used a neat client-server design so the Hylafax server ran in the background, and our fax script called the Hylafax client each time it wanted to send a fax. The client passed the fax onto the server, which queued it, and sent it when it could.

Hylafax automatically retried sending, a few times, so we only had to deal with hard errors (broken fax, run out of paper, hotel on fire, that kind of stuff). We recovered the status codes later, from the Hylafax log file, and pumped them back into the application.

It came down to a rather simple Bash script that took the next fax from an inbox (a directory of with one file per booking), called GhostScript to create a printable page, called Hylafax to send it, and then moved the fax to the outbox. It also created a log file that the application could read to know what happened. This script ran as a daemon, sleeping for a second if there was no activity.

It worked nicely. The good folk at Futuroscope were a little shocked that a simple fax modem and some magic software could replace the large industrial-sized fax machines their vendors were trying to sell them. Nonetheless, they saw it worked, shrugged, and the system ran nicely.

Only once did things stop working, when the application was moved to a faster, cheaper server. Turned out someone had implemented "sleep for one second" by busy-looping. On a multi-user system, no less. People had wondered why everything froze for a heartbeat when someone confirmed a booking. Yes, the code said something like,

```
PERFORM CONSUME-WHOLE-CPU
  2000000 TIMES
```

Their proposed fix was to increase the counter to 4 million times. I sighed and explained how to call the Bash "sleep" function from COBOL.

Lesson learned: identify the riskiest parts of your architecture and bring them under your full control. The point of using free software was that we could control everything using Bash scripts. We could test everything in isolation. We could in effect build a full solution, removing all risk, faster than it took us to learn and interface with existing solutions.

Successful use of free software in a commercial project was a shock to Sema in 1992. There was some talk of turning the fax gateway into a product yet I couldn't in honesty package a hundred-line Bash script for sale.

Wiring the Factory

Shortly after, and with UNIX, HylaFax, Bash working as mind-bleach for my UNIVAC PTSD, I worked with a Sema team on an industrial automation project. In short, this meant getting a large cement factory (owned by the firm CBR, one of the main Belgian cement producers) to talk, in a matter of speaking, with the SAP system that took orders and invoiced customers.

When we finished the project (and this is a flash-forwards), the head of our department came storming into our offices, quite angry. He showed us the figures for the work we'd done. We'd completely missed the budget and he was afraid it was going to cause real problems for the client.

I was a little confused at first because we'd worked hard and well, and pulled what was rather a mess into a coherent, well-working system. The managers on-site adored us, and so did the managers at customer HQ. What was the problem?

Turns out, Sema had made a fixed price bid, based on some random estimates of how much work we'd have to do, multiplied by the usual "engineers always underestimate the work" factor, plus random "project management" amounts, and so on. It was a fairly solid budget, for a major upgrade to the factory.

And we'd under-spent by such a large amount that Sema's finance department had red-flagged the figures as impossible. I laughed and explained that we'd simply done our work well. And indeed the customer never complained. We did not get bonuses, or raises. It was simply our job, and my reputation for pulling off miracles rose a few notches.

I'll describe the project itself, and how we got it to work, because it seems interesting. The factory produced dry cement, which was sent by truck all over Belgium. Clients would place orders, which would be scheduled, and then sent to the factory. Truck drivers (independent contractors) would come to the factory, get a shipment, and drive off to deliver it.

This was done largely by hand, which was slow and inflexible. Orders would be faxed through to the factory, where someone would schedule them, then enter them on a local application. Truck drivers would arrive, and at 7am the booths would open. In each booth, when a truck pulled up, an operator would choose an order, print it out, and give that to the driver.

The driver would then find the right loading bridge for the kind of cement, and give the paper to the bridge operator. He would control the loading pipes, confirm the order on his computer screen, and the trucker would go off to deliver it. Some trucks were dry, some got additional loads of sand and water and did what concrete mixers do.

Customers liked the cement, and hated the ordering process. It was slow and clumsy. If the weather suddenly got better or worse, there was no way to change an order to take advantage. You can't pour concrete when it's below freezing point. What do you do if there's a cold snap, and a truck full of the stuff turns up?

Industrial automation is a specialized business, and it was not ours. There were companies who knew this area, and one firm had built new automated loading bridges for the factory. Sema's job was to build a scheduling application. Another firm had built an application to replace the booth operators, and allowing more booths to be added over time. Yet another firm was providing various pieces of hardware. And then we had to connect the scheduling application to an SAP system (in Brussels, far from the actual factory) used for sales and invoicing.

None of this had been developed or tested beforehand. So when I arrived on the project, we had five different teams building stuff that was supposed to work together, and of course, did not. Apparently this was just standard practice: bring a bunch of stuff to a site and then hammer at it all until it worked. What made it extra fun was that each vendor had the attitude of "our stuff works, so if there's a problem it's yours, not ours."

It was made worse by the practice of fixed budgets. Each firm had made offers, which the client accepted. Spending an extra day fixing someone else's problems meant a day of lost income. All this is standard in construction projects. Yet building a complex software & hardware system is a different thing.

Let me give an example. The design used smart cards, a great idea. When a driver arrived, the kiosk would spit out a smart card holding all the order details. The kiosk would then say "Go to bridge 4" or whatever. At the bridge, the driver put the card back into a kiosk, the cement loading started, and that kiosk printed a receipt, for the driver.

So here we are with a UNIX system running the scheduling application, a PC running the kiosk app, and smart card readers. The team writing the kiosk app had no idea how to talk to the UNIX system, nor to the badge readers. No-one has experience with the smart cards, which the client has bought from Siemens. Siemens is not providing much support, and this is a decade before the web made it all easy.

And in every meeting, the attitude from each firm was, "not our problem, our stuff works." It was after several months of this toxic stalemate that I'd joined the project, in my usual dual role as plumber and fire fighter.

My colleagues and me adopted a simple solution, which was, "every problem in this project is ours." I figured out how the badges worked and we designed the badge data format. I wrote serial I/O routines to read and write badges, and handed them over to the kiosk teams. We built TCP/IP clients and servers to connect the kiosk application to the scheduling app, and gave the kiosk team libraries they could just call.

This all sounds like a lot of extra work yet we loved it, and were good at it, and it was faster and easier than arguing. At first the other teams were confused by our approach, and then they realized things were actually moving ahead. Little by little the project became happy again.

The kiosks were the cornerstones of the project, and were abysmally badly designed. The client provided the physical infrastructure, and the large metal housings. Various parts, such as cement-resistant keyboards, were ordered from afar. There were PCs to run the application, and little off-the-shelf Epson thermal printers to print tickets. Nothing had been tested beforehand.

And predictably, as the kiosk team tried to put this all together, nothing worked. It was almost comedic. The printers were not designed for kiosk use but as cash register printers. They had to be propped up at an angle with pieces of wood for the tickets to come out, and they jammed constantly. But the worst offenders were the screens. We'd started working in late winter, and by spring we were testing the kiosks for real.

As the sun rose, it dawned on us that the kiosk entrances faced south. The kiosk design was nothing as sophisticated as today's parking kiosks or highway toll kiosks. Flat panel screens were still science fiction. So here we have a PC in metal box with its CRT monitor behind a sheet of glass. The driver had to get out of their truck, their badge (which was both for ID and to hold the current delivery), choose a delivery, confirm it, and get their ticket.

With the sun shining from behind, the screen was unreadable. CRTs did not have great daylight visibility. In direct sunlight, behind a sheet of dirty glass, it was tragic.

One day, watching the frustrated drivers squinting at the glass as they tried to read the screen, I took a flat piece of cardboard, cut out a credit-card sized hole, and taped it over the glass. This fixed the problem. When I came back to the site a week later, both kiosks had metal plates, to my design, welded over the screens.

This was the tragicomic part. The project was also just painful in other ways. The client had specified that the kiosk application should be able to run even if the scheduling app did not work. This was sane, since the exchange of orders to and from the SAP system had never worked well. So if anything failed upstream, they could continue to ship cement, and then re-enter their orders afterwards.

The kiosk application thus had a "stand-alone" mode where, if it decided the scheduling app was not working, it would take control of things. Except, this app would detect a "failure" randomly, and the entire system would swing in to, and out of, stand-alone mode. Each time that meant fixing things up by hand afterwards.

It was stupid things like, the kiosk app would open connections and not close them, and exhaust file handles on the UNIX system. Or, it would run a 100% CPU background task that slowed everything down so that the I/O loop could not run, and timed-out.

There was a point where the factory would call us at 4am, saying the system was down, and we had to come on-site to get it working again. That meant a 2-hour drive. When we arrived, a long queue of furious truck drivers waited on us to fix things. Which meant restarting things, and manually consolidating a mess of local and remote orders. The real damage was done by switching in and then out of stand-alone mode.

The lesson here is that making systems more complex, to try to make them "reliable" will usually make them less reliable. What we were experiencing was "split brain" syndrome, the bane of any clustering design. Allowing the PCs to randomly decide they were in charge was

a Bad Idea. Letting them decide to switch back to normal mode made it much worse.

Since the crises were so random, we started sleeping in the factory offices so we could be there at 3am to try to catch it happening. To be frank, I didn't enjoy sleeping on a mat in between office furniture, in the heart of a massive industrial zone. Once we decided we'd bring beer, it was slightly better.

What we eventually came to was a panic button on the PC screen that forced stand-alone mode. If orders stopped arriving from the UNIX system (for any reason, one does not care at 5am), the operator clicked the panic button, and the PC switched to stand-alone mode. The operator could then handle the morning peak, with a little more manual input. Somewhat later when things were calm, we'd consolidate orders with the UNIX, investigate what had caused the fault, and switch back to normal mode.

Additional lesson: fail-over automatically and recover manually.

Despite all this, we finished our work so far under budget that it raised alarm bells. The client loved us, having seen us in action. Years later, this would pay off.

The main lessons I learned were obvious and yet worth stating, perhaps because the industrial world is so distant from the normal pure digital development world:

- Don't expect random work from random vendors to magically work together.
- Do not develop and test in your production environment.
- When things go wrong that even marginally involve you, assume responsibility until you know where the real problem lies
- When you use hardware and software that bends to your will, you can work significantly faster.

Fear and Loathing in Brussels

In an industry that is driven by forwards change, it is shocking how many projects are based on betting against the tide. By this I mean forcing an application to run on some antiquated, expensive, inflexible, and painful platform, when new, cheap, flexible, and enjoyable alternatives exist.

It almost always ends in disaster for the organization. And yet I've seen this happen so often that it is almost a caricature. Brain-dead management forces obsolete technology on team who build mediocre application that fails to fix organization's real problems, which then gets taken over by smarter competitor.

Here are some of the reasons this happens so often, in my opinion:

- **Psychology of sunk costs**. "We've spent twenty million on our goddamn mainframe and goddamn you if you think we're going to junk it just because you have this new fancy-wancy UNIX thingy."
- **Self-interest of vendors**. "Sure we can expand the main memory from 64MB to 128MB. It may be a little expensive, but mainframe memory is special! It's better than that cheap, unreliable UNIX memory!"
- **Politicization of IT**. "My departmental budget is $10M per year, and you're talking about slashing that by half and yet adding 10x more users? Are you insane? You're fired. I'll find a consultant who agrees with me."
- **Fear of the unknown**. "Why would we use that 'UNIX' thingy? It's just a large PC, totally unreliable. And anyhow, UNIX will never be widespread. No, we prefer our tried-and-tested mainframe architecture that runs real networks like SNA LU6.2!"

Shortly after the UNIVAC trip, I was thrown into one more such project. The team was building a new financial system. The team was using our COBOL toolkit. They were using AIX UNIX machines for development. Production was to be on a BULL TDS system. One. More. Obsolete. Mainframe.

The entire project was designed as an internal hostile takeover of the firm. The firm was a recent fusion of two businesses. The profitable one consisted of many smaller offices, each working their own way. Clients got customized service and paid a premium for it. The unprofitable firm however had the power, for political and historical reasons.

So the new planned system would enforce order and consistency on those cowboy offices. It would in effect break their independence and bring them into a neat, centralized organization. It was a classic battle between good and evil, between sanity and insanity. And once again, I found myself on the wrong side of the battle lines.

The architects of the new system had no idea, really, what they were going to build. They could not ask their users, because they were at war with them. So they made it up as they went along. Designs and plans fell out of meeting rooms onto the developers' desks.

The developers were decent people. This was the era when even with modest training, you could learn to develop business software. We had as many women as men in the crew. I cannot fault the developers, nor the COBOL language, for the crap that emerged. This was a direct result of getting insatiable, arbitrary demands from the analysts.

My job was to support the developers with source control and continuous builds, and to build the technical framework to make the applications run on TDS. Bull's computers were feeble imitations of IBM's mainframes, and likewise, TDS was a feeble imitation of IBM CICS, the dominant mainframe transaction processing system.

Lurking like a bridge troll underneath the application was an Oracle database. I won't comment further except to say that it more or less worked, yet was both a major source of problems, and apparently, a major slice of the project cost. We could have done much better IMO.

In those days the only plausible source control system was CVS. We did not use that. Instead we used shell scripts to check-in and check-out sources from a repository. The application consisted of hundreds of programs, each was either interactive (basically, one or more screen pages with logic), or batch.

When you checked out a program you got all its related files. You could edit, compile, and test locally, quite quickly. When you checked your program back in, we saved the previous version (saving the differences, to save space), and stored the new one. We compiled everything, once per hour or so, to create a fresh testable version of the application for test users.

As new programs got checked in, they got sent to the Bull system for test compilation. That produced error listings (the code trying things that worked on AIX yet not on the Bull). Our shell and awk scripts pulled these error reports back to the AIX, extracted the few lines of error message from 20-page listings, and passed them to the developers so they could fix things.

Some of the developers (especially the older ones who had learned the Bull environment) just printed out the reports, then stacked them in heaps around their desks. These heaps grew higher and higher, eventually, forming walls behind which the developers hid.

The test users marked changed programs "approved" when they were happy with them. To make this work we had developed our own issue tracking system, in COBOL. That was not hard, as these tools worked well and let us build applications rather quickly.

When a program was approved, it was sent up to the Bull system for real this time. It was compiled, and from then on available to external test users.

People noticed quite early that the AIX test environment was rather faster than the Bull. Out came the arguments about interactive sessions vs. transaction processing. "The Bull will scale better for hundreds of users," went the official line. No matter that even with a handful of users, it was already slower. They added more of that special mainframe memory, and upgraded to the largest CPU possible. Using the Bull was not negotiable.

Our development environment was quite neat, I think. For developers, it all just worked. Of course I far prefer our modern git-based environments. Yet what we built showed the power of straight-forward UNIX shell scripting.

We also cracked the TDS environment and got the application running on it. This was no simple job, mainly because the programs had gotten so large that they did not fit into memory. Like a lot of machines bypassed by history, real memory was larger than virtual memory. That is, TDS limited programs to ridiculously small sizes, while the actual system memory was much larger.

You could even access the system memory from COBOL, just not portably. And we depended on making TDS and AIX look exactly the same from the point of view of developers. Which ended with me writing a memory manager in COBOL that managed system memory, swapping blocks of memory in and out of application space, behind the scenes. It very fast because the heavy work (copying memory) compiled down to fast machine code. If in COBOL you say "move these 30,000 bytes from A to B" that can be a single machine instruction. If you write a loop, moving 30,000 bytes one by one, it adds up to maybe a million instructions.

No matter, we could not save the project. The developers went through death marches, the regional offices went on strike, managers left and were replaced. By the end of 1995 the client had stopped trying to blame Sema and our team for the failure. Everything we did worked. The project was postponed indefinitely, and I went off to find more constructive things to work on.

Welcome to the Web

In 1995, the entire World Wide Web fit into a single book, with one paragraph per website. In November 1995 I registered imatix.com and started to think about building an Internet business.

I'd been working in my free time and weekends with my friend Pascal on web servers. The design was inspired by the server I'd built for that tour operator. You can handle a lot (a *lot*) of sessions in a single process, if you use what's called a "cooperative multithreading" model. This means each session does a little work and then gives control back to a central dispatcher.

Threads must never block, and all I/O must be asynchronous, just as on the VAX with its AST calls. You don't need actual system threads. That's nice because real threads bring a lot of nasty, often fatal problems with them. We've learned by 2016 that sharing so-called "mutable" data between threads is a Really Bad Idea. In 1996 I'd less data for this, yet already knew that it was nice when a thread could work with global data without risk of stepping on other threads' toes.

By December 1996 we released a working web server, which we called "Xitami." Xitami was one of the first free web servers to run on Windows. Microsoft's "personal web server" was crippleware, and people hated it. Xitami was easy to install, it was fast, and had no limits. You could, and people did, handle enough traffic to swamp a highest speed Internet connections.

I once saw a Slashdot article about a guy who'd hooked 26 hard drives to his Windows 95 box. He had a web page showing the PC and explaining how he'd done it. Slashdot was one of the most popular geek news sites, and the name had become to mean, "kill a website through sheer volume of requests." He was running Xitami on this PC, and it didn't crash or slow down.

Yet I'm not going to talk about Xitami. It was my first popular open source product. It won many awards and users absolutely loved it. Yet it made us no money and had no developer community, and got us no business. As I define "open source" today, it was a failure.

Our real product, on which I wanted to build a business, was something quite different. I'd designed a "web transaction protocol" and implemented that in Xitami, together with some tools for building web forms. This gave us a crude yet working transaction processing system for the Internet.

That is, by 1997 or so we were able to build usable web applications that could handle thousands of concurrent users on cheap hardware. HTML was really poor. Basic functionality such as "put the cursor on this input field so the user can fix an error" was missing. Yet it worked.

In that company with the Bull TDS there was a separate, independent division building their own applications. They'd been searching for some way to provide access to remote users. They were well aware of the potential of the Internet, even if that mostly ran over dial-up modems. They took one look at our web framework (which I called "iMatix Studio") and asked when they could start using it.

Studio wasn't trivial to use. We'd not spent any time looking at developer languages, and we had totally missed Python as a candidate. Java 1.0 was just released, unstable, slow, and unusable. We did not look at more esoteric languages, and I was firmly against C++ because it produced such fragile code. So, we developed in C, which is less than ideal for the grunt work of business applications.

Still, it worked, and we slowly built up our framework. We lacked good code generation, so we built a generic code generator called GSL, which we still use today in the ZeroMQ community. We lacked a language for modeling the state machines and UIs we wanted to generate, so we first wrote our own structured data language, and then switched to XML by mid-1997.

It was in 1998 that I decided to part ways with Sema. They paid me well, and they were decent people, yet they were consistently betting on the wrong side of history, and it became deeply troubling. The last straw for me was when my boss asked me to make a technical design for a nationwide insurance network.

Our brief was to connect several thousand insurance agents in a network. They would log into a system, query insurance files, get quotes, log events and claims, etc. It was almost exactly the same problem as our old tour operator system, seven years down the line.

So I made a design betting on the future. We'd build it as a transactional web application, and use a number of XML formats for document standardization. The framework would be Studio, using arbitrary developer languages. We'd run on a single large UNIX box, which we'd tested and shown capable of dealing with ten times more traffic than planned. We'd use an Oracle database (again, customer decision, not negotiable).

Sema presented this design to the client, while simultaneously presenting a second design based on a client-server framework called PowerBuilder. At the time, at least in Belgium, PowerBuilder was fairly popular for such wide-area applications. Microsoft supported it and promoted it. It was expensive and produced a lot of money for vendors like Sema simply by giving them a slice of license income.

The PowerBuilder user interface was also much richer than HTML, at the time. It was however a miserably nasty system to deploy. When you started your app every day, the first thing it would do is download updates to the client half. This could take five minutes, or it could take an hour. You could not do updates during the day. Users *had* to log out at night.

Yet we were talking about a system that was to be developed gradually over a period of years, and was meant to last for decades. The goal was to define new industry standards, and to bring the insurance industry into the modern world.

And my design was rejected on the grounds that it could never work, that dial-up was no basis for a serious application, and that the Internet was just a toy that would never come to anything.

I shrugged, and went and started iMatix Corporation sprl. I started to plan, with my friend Julie, how to build a real business. It was all new to me, so we kept it small and modest. Sema kept asking to help save catastrophic projects, and I kept accepting. We needed income.

We worked on a project for the Belgian railways. Again, a bet against history, with a massive distributed application running on VAX/VMS. No transaction processing with ACMS. No attempt to reduce costs. Instead, use a platform that was already dying. Compaq had just bought DEC, the producers of the VAX, and would slowly start killing it. I stayed only a few weeks, then made an excuse and left.

We worked on a death march project for the Francophone school system. Once again, a bet against history, with production data on an IBM mainframe, and a nasty client-server product that depended on Windows NT servers in every school. Once again I proposed a web based architecture to solve the interminable woes of this over-complex architecture, and was asked to back off and stop being disruptive.

It was in 1999 that Manpower International in Brussels asked me to help on a small project, which turned into one of the most widely-used and most successful web applications Manpower had (and perhaps has) ever built.

UltraSource, The Hot Potato

FYI, Manpower's business was to select and recruit staff, and then send these out on temporary work for clients. It is a business called "sourcing", and more specifically, "HR sourcing" that I got to learn in some detail.

Manpower had a core IT strategy that was quite common at the time, in larger multinationals. That was to build a central information processing system, and then use that everywhere. For a company like Manpower, that meant using the same application for every customer, in every country.

It was the same strategy that sold products like SAP, aka "liquid cement." It is a strategy that is especially seductive to managers trying to build a company through acquisition. Buy a firm somewhere, throw out their existing core IT systems, replace with whatever corporate standard the last CTO fell for.

As a business strategy, it can work. Modern banks have relied on this because otherwise their separate acquisitions simply can't work together. Modern firms like Google, Microsoft, and Amazon each rely on their own global, shared infrastructure. Yet banks and Internet firms are special cases, where sending data around the organization is their core business, rather than a problem of profit & loss reporting.

In the case of a firm like Manpower (as with that firm trying to move all its offices to a shiny new Bull mainframe application), it was a rather insane strategy. Manpower's competitors tended to grow by acquisition and then leave every business unit to do its own thing. That was far more organic, and cheaper, than trying to rationalize all their IT systems.

When we arrived on the scene, Manpower International (which had the task of providing software applications to the whole world of Manpower companies) had tried and failed miserably to build a global sourcing application. We did not know this, of course. The problem, a classic hot potato project that no-one wanted to touch, had fallen back in the laps of a small team in Milwaukee.

Some time later Milwaukee asked me to make a proposal and so I flew there to spend a week talking with a room of 20-30 key people. They were experienced, skeptical, and critical. I brought with me a small prototype of some core functionality, an order placement screen.

It didn't happen overnight. Our contact in Brussels had spent some months funding us to make the demo, and prepping the room in Milwaukee. Her home office was Manpower Japan, and that company was having a hard time getting larger clients. Her vision was to build a *real* sourcing application that would solve these difficulties. She wanted, ideally, the Americans to pay for the work.

When we started talking about large scale web applications, my heart leapt. Finally, a project that aligned with my vision for iMatix. And then the reply came, "sorry, company standard is Microsoft Windows, Microsft Transaction Server (MTS), and Microsoft SQL Server"

Now Windows is just a box and our Studio software ran fine on it. Indeed it was on Windows that Xitami got the most love from users. By that time we were already running Linux heavily internally, for our SVN source control, our mail server, and so on. It was clear that Linux was the way forwards for servers. Yet Windows would make a fine interim step.

MTS however, was a WTF? It was a poorly documented, fragile mess. From our first tests we found that it crashed if you smiled wrongly at it. It was unscriptable, so all administration had to happen via point and click menus. Applications would freeze randomly. It did not play well with SQL server, so we got deadlocks and timeouts. It was a black box with little visibility on what was going wrong.

I argued all of this, and pleaded to use our own tools. The answer was no, no, no. So, we wrote the prototype in Visual Basic and then somewhat later we (myself and our Brussels manager) found myself explaining it to a roomful of people in Milwaukee.

It seemed like a rather hostile room. One or two of the people were sympathetic. The rest seemed truly unhappy to be there. I didn't take it personally, and just continued with diagrams of architecture, the tools we were making, walk-throughs of the code, and so on.

And a few days after we flew back to Brussels, word came that we'd gotten the project. Turns out, absolutely no-one wanted to touch this project except a few people in Milwaukee. This small core of 4-5 people had long experience with HR sourcing, and successes under their belts. They saw the potential in our proposal and they realized they could make it work.

The foundation for a good project is: a competent client who knows the business and has power of decision; a full-stack team that can deal with the work, at all technical levels; and a technical platform that is both dependable and tractable.

In the first real meeting we had with the core team, they showed us a large stack of designs and ideas. I looked at them, then made a counter-proposal. Let's take the prototype we have, I said, and improve it little by little. You make requests, we'll make changes, you test the results. How often can you make new releases, they asked us. Every day, if you want it, I replied. We agreed on two or three times per week.

And so we worked, with three people taking our work, testing it, and coming back with new requests. We'd make ten or fifteen changes a day, test them, and push them to the test machine. We logged the work in a Jira issue tracker. Every now and then we'd have face-to-face meetings to work through difficult problems, to break them down into small pieces.

Windows remained a real headache that we could not fix. Ironically the person who'd insisted on this left Manpower not long after we'd started, and no-one else cared. Yet it was too late and we were stuck on that. If I'd known how much leverage we had before we started, I'd have insisted on using our own tools.

On top of Visual Basic and MTS, we built our own framework (iAF, for iMatix Application Framework), a powerful code generator that built web pages, object managers, and database layers automatically. It worked well and we used it in other projects. Yet it was always based on that shitty toy scripting language and that nasty imitation of a real transaction processor. And so it never gave us real value.

This was perhaps the biggest mistake I ever made in my business. The lesson here is, when you are a technology company, use your own technology in your projects. Don't accept client requirements that sabotage your own vision and future. It isn't worth the short term gains.

Much later I came to frame this more brutally as, "never build closed source, period."

iAF used meta languages to describe the pieces of the application. Later I'd call this "model oriented programming." We defined a *presentation layer* consisting of screens, an *object layer* consisting of objects and views, and a *database layer* consisting of database tables and indexes.

Here's a simple example, for working with currencies. The database layer defines the actual properties we store for a currency:

```
<table name = "currency" description = "Currency" >
Currency lookup table.
    <field name = "isocode"   domain = "currency">ISO currency code</field>
    <field name = "name"      domain = "longname">Currency name</field>
    <field name = "showcents" domain = "boolean">Show cents?</field>
    <field                    domain = "audit"/>
    <index name = "primary">
        <field name = "isocode" />
    </index>
</table>
```

We then specify an 'object' that inherits from the database table, and we add some flair (such as here, the name is mandatory when creating or updating a currency):

```
<object name = "currency">
    <require>
        <field name = "name" />
    </require>
</object>
```

Behind the scenes, iAF bases that object on a 'default' object that looks like this (this is a standard model that the developer doesn't need to know about, or touch):

```
<class name = "default" default = "1" >
    <!-- The create view is required by the object layer       -->
    <view  name = "create"  read = "0" write = "1" delete = "0" />
    <!-- The delete view is required by the object layer       -->
    <view  name = "delete"  read = "0" write = "0" delete = "1" sameas = "create" />

    <!-- These views are recommended but not obligatory        -->
    <view  name = "detail"  read = "1" write = "1" delete = "1" sameas = "create"/>
    <view  name = "summary" read = "1" write = "1" delete = "1" sameas = "create"/>
    <query name = "detail"  view = "detail"  />
    <query name = "summary" view = "summary" />

    <state name = "object exists">
        <view name = "detail" />
    </state>
</class>
```

And then I describe how I want the object views to appear to the user. At this point I start mixing Visual Basic code with my model. The code generator includes my custom code in the generated result. Complex screens have thousands of lines of custom code. Simple ones like this, just a few:

```
<screen object = "currency" style = "select" alpha = "1" />
<screen object = "currency" style = "list"   alpha = "1" />
<screen object = "currency" style = "create" />
<screen object = "currency" style = "detail" />

<macro name = "currency_validate">
<use object = "currency"/>
<handler event = "on_accept" >
    fld_currency_isocode = ucase (trim (Request.Form ("currency")))
    <if condition = "fld_currency_isocode &lt;&gt; &quot;&quot;">
    <fetch object = "currency" view = "summary">
    <missing>
    cur_message = "Please enter a valid currency code or select from the list"
    cur_error   = "currency"
    </missing>
    </fetch>
    </if>
</handler>
</macro>
```

The application architecture isn't complex. It's 1999 and we're aiming for consistency and simplicity, not features. It is monolith built on a single database. There are no asynchronous updates to the web page, no AJAX. We use JavaScript for local validation and cosmetics, such as flagging an error input red, and putting the cursor there.

What we built turned out to have some interesting aspects:

- It was a large application, with a hundred database tables and five hundred screens. With our tools we were able to build new functionality rapidly.
- It was slow to use because often it took lots of clicks to get to a certain place. We did not spend much time on creating fast paths or shortcuts.
- Users loved the application.

The thing was, users did not work 24/7 on the system. They used it for perhaps half-an-hour a day. Some users (especially inside Manpower) used it full time, yet they were happy too.

The reason was simple in retrospect. This was an application meant for clients of Manpower, thousands and thousands of HR managers in random firms. The number one expense for Manpower, in previous projects, and the number one fear of users, was training and complexity. What we built was so consistent and simple that you could use it with zero training (of course, you had to know the business).

For instance we used a list/detail design for working with data. You clicked on 'Currencies' and got a list of currencies. Click on a currency and you see its detail. Click 'Edit' and you can change it. And so on. Learn this design once, and it worked almost universally across the application.

Obvious, and yet hand-built applications simply did not work like this.

Occasional users cannot learn complex UIs. And when they get confused they ask for help, and that will overwhelm any support structure you can build, and be ruinously expensive. And *this* was the reason no-one wanted to take on the hot potato.

The lesson here is a devious one, and that is that you often don't know what the real problem is until you get very close to it. We were lucky our approach fit Manpower's way of working.

We eventually rolled the application out to Japan, the Netherlands, Germany, the UK, and the USA. This happened gradually, over years, without major stress. We built a translation tool that let local offices translate the screens. We added customizable order pages so that local offices could make orders as baroque as they wanted.

The system wasn't perfect, in many ways. We had bugs here and there that only showed when the system got stressed. There was an order export process which pushed completed orders out to other applications for processing. This was long before we understood how to do messaging, so our designs were clumsy and not fully robust.

Above all, MTS would panic and shutdown threads when too many users worked at once. There was no way to fix this. It left the database with dangling locks that ended up killing the entire system. We had to limit the number of users, and move to larger machines.

Despite these stumbles, the application ("Ultrasource") became Manpower's first global web application, and ran for many years. It produced vast amounts of new business for our client, and gave iMatix a healthy income in ongoing maintenance fees.

And then in the summer of 2000, I got an email from a Nigerian beer company (Nigerian Breweries, or NB) asking if we had any experience with electronic payments systems, and could I please read the attached documents.

E-Payments in Lagos, Nigeria

East Africa was my first home and I'd traveled many times to the continent, for family and work. Sema would send me on short trips to do UNIX trainings, help with EASY installations, and so on. I'd been to Burkina Faso, Togo, Rwanda, Angola.

Either you love Africa or you hate it. There is no easy middle ground. Even stepping foot in Africa as a European is a political act, conscious or not. It took me many years to understand it as a continental prison filled with innocents, as I've written in my book Culture & Empire.

So the prospect of a project in Nigeria was appealing. I read up on the place in Lonely Planet and a book called "The Most Dangerous Countries in the World," which ranked Nigeria just after Columbia. My mother pleaded with me to not go there.

Not being a fearful person, I read the client's requirements, and started to work on a project proposal. It was an interesting case and I realized why they'd come to me. They used EASY for their accounting and sales. They had to extend that with some kind of a network that could carry payments to banks. We'd started to get experience connecting bizarre systems together, and building successful web applications.

NB's IT manager had asked Sema, "who do you know who could possibly make this work?" and they had replied, "ask Hintjens, he's probably the guy you want."

Let me explain what the problem was. NB was doing well. They dominated the market, and Nigeria was (and still is) a huge market for beer. The profit per bottle was low, yet sheer volume made up for that. NB was expanding, building new breweries across the country. They had powerful marketing and sales, and frankly, their beer was excellent.

Two things struck me, when I first visited. First, the brewery was perfectly organized. The buildings were all in good state, the gardens neat and tidy, the production line modern and shiny. It was a striking contrast from a brewery I'd visited in Kinshasa some months before, where crates of broken bottles lay around randomly, where people seemed tired, and where a heavy feeling of tropical lassitude lay around the place. NB's facility hummed with positive energy.

The second thing was it was run entirely by Nigerians with some other Africans. There was just one European, the financial director, or FD. Later, as the brewery upgraded its capacity, teams of eastern European engineers would fly in. And there was our team, as the project progressed. Yet in my first visit, I don't think I saw a single white face from leaving the airport to departure, two weeks later.

The IT manager took good care of me. We visited Lagos, a massive and busy city filled with life, most evenings after work. I've spent a lot of time in Africa and was rapidly at home, no matter where we went. After a week of design discussions and meetings with other managers, I went back home to write up a detailed proposal. I can still remember my shock, in the airport departure lounge, to see white faces and feel, "how strange they look!" and then realize.

Anyhow, back to the problem. Nigeria, at the end of the 20th century, was a huge and booming economy driven almost entirely by cash and favors. Much of the country's wealth came from oil, yet there was (and has been, in this part of Africa, for hundreds of years) a solid commercial middle class. Spend a day in Lagos and you see the sort of frenetic hustle that feels more like New York than Atlanta.

And yet, as the FD of the brewery explained, there is no system of credit. A "long term" bank loan is 3 months. No credit cards, unless you're a foreigner in a foreign operated hotel. No checks. Few people have bank accounts, and when they do, it's for moving hard currency in and out. Salaries are paid in cash. Cars and houses are bought and sold in cash. There are no mortgages, no car financing plans.

The brewery produces beer, which it puts into glass bottles. These go into crates of 12 or 24, which get stacked onto lorries. The lorries then take the beer to distributors. The smallest unit of sale, for the brewery, is one lorry load. The distributors then resell the beer to shops, clubs, bars.

When a lorry loaded with beer leaves the brewery, the distributor has paid for the beer, the empties, and the crates. The lorries belong to the distributor or their transport firms. The brewery does not own and operate its own trucks.

And the distributor has paid for the liquid beer, the glass bottles, and the plastic crates, in cash. One lorry of beer comes to, let's say, two large suitcases of cash in the highest denomination notes. The brewery staff count and check the money before the lorry leaves. The cash is then put into the treasury, a large room that is literally filled to the ceiling with notes, at times.

When a lorry returns, the brewery counts and checks the crates and bottles, and the distributor gets repaid (in cash) for the value of the empties. Remember that this is rather more than the value of the liquid beer. This means the brewery is, at any time, holding a lot of cash simply as deposits. It can keep some of the deposited cash in a bank yet a lot must remain on-site.

The backdrop to this successful business is a currency (the Naira) that continues to fall in value, so that the stack of paper needed to pay for one lorry keeps getting larger.

Then, a culture of criminality that is pervasive and can be shocking to foreigners. We're used to trusting others around us, and we're shocked when someone steals a wallet, or a car, or a phone. In Nigeria, there is no trust. All business runs on the assumption that theft will happen if it can. Entire lorries of beer have disappeared, never to make it to the distributor.

Then, a severe lack of technical infrastructure. In 2000 there was a fixed phone network that mostly worked, though only wealthy people and businesses had a phone. Electricity would fail several times a day, so everyone who could afford them had back-up generators. There were some good highways yet most roads were miserably bad, and jammed with traffic.

Driving around Lagos was the fuel of nightmares. You'd see a flame ahead on the motorway, it was a burning truck tire that someone had left in the road. Why? To mark a hole large enough to fall through. People would carry extra fuel in their car boot (since fuel supplies

were so sporadic). So if they were hit from behind, their car would explode. Taxis and scooters ("okadas") drove maniacally around pedestrians, goats, street vendors.

After dark, there were roadblocks with armed police. I don't really know what they were looking for. We just greeted them, said we were from the brewery, and they waved us on. This happened so many dozens of times that I'd greet them with a huge smile, a handshake, and "shine shine bobo!" which is the slogan for NB's most popular beer, Star. They'd laugh and we'd drive on.

I'm rambling. Back to money. The question was, how could we cut out the cash transactions at the brewery?

After some thought, our solution was to have transactions happen at the bank side of things, rather than in cash on-site. So each time a distributor bought a truckload of beer, they'd transfer money from their account to the brewery's account. And each time they returned an empty truck, the deposit would flow in the opposite direction.

It sounds simple and obvious enough. Yet bear in mind, we can't trust individual bank employees, nor do we have any kind of remote (web based?) banking system.

Our first problem was to convince the banks to work with us. We learned quickly that a meeting for a certain day meant, "we'll arrive at some point during the day." Traffic jams could last several hours, even though from the brewery to Victoria Island, and the financial district, was just 15 minutes' drive.

The banks at first thought we were somewhat insane. No-one had ever suggested electronic payments seriously. As thought experiments, sure. For real, live business use, let alone for considerable amounts of money, they were skeptical. So we explained the model, which was based on a secure messaging system running over pure old email.

This took us some time to figure out. First we hooked into the accounting system, which was our old friend EASY. First we ran a continuous background job that caught orders flagged for electronic payment. This was simple to do. These orders were sent to our app, running as a web application on a local Windows server. A manager would get an email alert and click the URL to sign in. They'd review and approve the order. That would then go to their boss, who would also review and approve the order.

Once approved, a payment instruction would go to the bank. This tells the bank the accounts to pay from, and to, the amount, and a reference. We signed and encrypted the payment message, then sent it to an email address that the bank used. There were five or six banks, as distributors used their own banks.

Each bank ran a Windows server with the app on, but when they logged in they saw payments, rather than orders. The app would fetch email continuously: dial up to the ISP, log in to the POP3 account, fetch email, delete email from the POP3 account, wait for five minutes, and repeat. As payment instructions arrived they'd be loaded into the app database and be visible to the bank.

We looked, in each bank, at integrating with their systems. That turned out to be infeasible. These systems were all different, and all bizarre, and the expected volume of orders was not so high (a dozen a day per bank).

So we agreed that the bank staff would simply make the transfer by hand on their own systems and confirm it on our app, when done. This left scope for fraud yet no more than normal inside the bank. Banks could, and did, have their own internal approval systems for transactions above a certain size.

Once the transfer was made, a payment confirmation would be sent back (again, by encrypted email) to the brewery. Our app would receive this, flag the order as paid, and tell EASY.

EASY had no existing way to import such data, so I wrote a small tool that acted as a TELNET client, and could log into the application and push the right buttons. Sema were quite surprised when they realized how we'd done this. They were expecting NB to pay for work to make a new import program.

Corinne (perhaps the fastest developer I've ever worked with in my life) and I designed the app and she did most of the development. Pascal helped with the backend, as he'd done for UltraSource. It ran nicely. By this time we knew our tools well, and had built other apps with them, like our own issue tracker, ChangeFlow.

Somewhat to my surprise, we were able to deploy the system in test, and we started to expand it to other breweries (NB had seven or so, in different cities). The amount of skepticism was massive, both in banks and in the brewery itself. Distributors -- who suffered the most under the cash based system -- were enthusiastic. To test, we sent payment requests to banks, reconciled the responses, and checked that things didn't get lost. At the bank side, the responsible manager simply clicked "Done" without actually making a transfer.

Dial-up in Nigeria was fragile. There are lots of failed attempts, busy lines, dropped lines. It might take half a day for messages to start getting through. If a driver had been waiting for their order to clear, they'd wait half a day. That happened with cash too, as it might take some time to assemble the needed stashes of notes. One thing you quickly learn in Africa: patience.

Yet apart from that, messages did not often get lost. I learned that email is surprisingly robust, even though it has no delivery guarantees. We used a simple retry mechanism to resend messages if we didn't get a response within some timeout. We ignored duplicate requests and responses. And so on, the usual stuff.

Lesson here is: you can do a lot with little, if you are creative.

Then just a month or so before we were going to go live, Heineken bought NB (they were already a minority owner, then they bought more shares, to get a controlling stake). They stopped all investment in EASY and the electronic payments system and began to plan a SAP deployment. And that was that. We packed our bags, and went home.

Building the Perfect Kiosk

In 2003-2004 we rebuilt the delivery system for CBR's cement factory in Gent. This time we got the full project, down to the kiosks.

We'd done a good job with the previous automation project, so CBR called us in to one of the earliest planning meetings for their new factory project. They drew a schema of all the pieces. We had, as before, one supplier for the kiosks, one for the loading bridge automation, one for the Unix application, etc.

By this time CBR were talking to us (iMatix) directly, rather than Sema. In the meeting I stood up. "Last time, we had real difficulty integrating all these pieces," I told the managers. "So what do you suggest?" they asked me. I took the pen, and drew a large box around all the pieces except the loading bridge, which was out of our competence. "We'll do all these," I said. There was about two seconds' silence and then the project lead, who still adored us from the work we'd done, said "OK," and that was that.

I did what I often did in those days, when starting a new project. I sat down and wrote a design spec.

For once, and because we knew exactly what we had to make, and why, the design spec was almost perfect. My main goal was full off-site testability, for every piece of software and hardware, alone and together. I also wanted to make the kiosks completely fool proof. Plug and play and never break.

I asked CBR what their budget was. We agreed on a budget for the software on the usual lines: days times rates equals total. For the hardware, we agreed on unlimited budget, with no profit for iMatix. This freed us to find the very best hardware. The kiosks turned out *very* expensive to build, yet considering the cost of failure, and the overall project cost (automating the deliveries for a major factory), this was easily justifiable.

Julie and I designed the hardware by looking for all the smallest, most resistant pieces on the market. Sun-readable screens, dust-resistant printers, badge readers, PCs. We designed the casing and interior layout ourselves, and found a firm to build six metal housings. Julie found a paper supplier and got a palette of custom thermal paper produced.

Let me tell the printer story as an example. We were discussing with CBR about the tickets the kiosk should print, size and type of printer. We agreed, thermal printers, no ink to smudge or replace. I asked if they had a preferred supplier, and they did. So we asked the supplier what kind of printers they had. The smallest was the kind you see in an airport, about 30cm high, 75cm deep and high.

"It has to fit inside a small box. Got anything smaller?" I asked. I had made this sketch of the internals of a kiosk design, and the printer had to fit in the space of a small stack of paperbacks. "No," they replied. "I'll find another supplier on-line," I told them. "Good luck, it's impossible!" they replied, not entirely sweetly.

We did finally find the right printer, tiny and fast, designed for fitting inside a kiosk. So the eventual kiosk was about 75% dedicated to holding paper, which was perfect. The kiosks could run a week without needing a refill.

Lesson here is, don't take "impossible" as an answer. Everyone lies, not always deliberately. We just have our assumptions and ignorance and we believe we're telling the truth.

Mato designed a multilayer Linux OS that booted off DHCP, and then fetched its application from the network and then connected to the server. The kiosks were plug-and-play: connect power and Ethernet, and they'd boot in about 10 seconds and show their welcome screen. Access control to admin functions on the kiosk was via special badges and PINs.

Thierry and Pascal wrote a new dispatcher, and I designed XML messages that connected kiosks to this backend system. Jonathan and I wrote a new reliable messaging system (STEP) that talked to the central SAP system.

Ewen and I built the kiosk application; I designed the twenty or so screens in black and green, using a large sci-fi font and movie-style computer graphics that were bold and easy to read even in direct sun. Ewen brought them to life, with his code talking to the dispatcher over the network, using a simple TCP/IP protocol.

In total eleven people worked on this, in offices around the world. We tested each piece, and each kiosk separately, without setting foot in the factory. The client build kiosk housing, took our work, installed it, and it all ran first time.

What else do you expect?

There's a review of the project that I published rather later that gives more details.

There were some good lessons here:

- Build up trust with the client and sometimes they will reward you for it.
- When you've paid for all the mistakes, you should know how to do it right the next time.
- A good specification lets diverse people work together without confusion or conflict.
- If you can test each piece alone, and you have reliable ways of putting them together, the whole should work.
- Don't be afraid to charge the real cost.

On the downside, the kiosks worked *so* well that the client never came back for support or maintenance. While this project made us money, it did not lead to any new business, and did not push my vision for iMatix forwards at all.

In that sense, it was a total failure. It is ironic that a "successful" project can be a failure, while catastrophically bad projects can push you through to better things.

I also learned that building a team just to have a team was wasteful. Employees take time to manage. I was becoming a middle manager, and not coding any more. We tried extremely hard to find new clients, and built several potential products:

- A HR sourcing application ("Sourceflow") inspired by UltraSource. We rebuilt the whole core and UI and database. We'd made numerous apps using iAF by then, and Sourceflow was elegant and nice to use. We showed it to many large businesses and HR suppliers. Lesson learned: don't make stuff and then try to sell it unless you are growing an *existing* client base. Sourceflow went into the trash.

- A plug-and-play kiosk design for factories, airports, carparks. In 2004, this was still a new thing. We had excellent software, and what I think was a nice hardware design. We did some sales work. No luck. Into the trash (luckily it was just a paper design). Lesson learned: breaking into markets you don't know is probaby impossible.

- A group-chat-as-a-service application called SMS@. You created a "site" using the sms-at.com website, and then people could use it via their mobile phones. We deployed this in Belgium and sold it to TV stations, and events like Brussels Rollers (people subscribed via text message and got news back, about cancellations etc.) SMS@ was really neat and worked well. However we had to pay so much to the mobile phone operators (2,000 EUR/month per operator just to be connected), that we needed to charge the users per SMS. I wanted much cheaper text messages but the operators were pushing for premium messages. Lesson: mobile phone operators are crooks who steal billions, fifty cents at a time. I finally killed the product.

In 2004, the IT industry in Belgium was still in crisis and though we spent a lot of effort and money on marketing and sales, we could not sustain it. We simply could not find new clients. Years of built-up cash reserves were draining away. One by one I fired my team, until it was

just a skeleton crew (Fabio and myself) left.

It was terribly sad to walk through our offices, where fifteen people had once worked, to see one or two people there. Yet without shutting down our projects and going through the pain of firing friends, iMatix would have gone bankrupt.

Lesson: be aware of your expenditure and manage your losses. You can survive a long time with less income if you are in tight control of what you spend.

Second lesson: it is no favor to pay people to do idle work. When you hire someone, tell yourself, and them, one day this will be over. Today I far prefer working with self-employed partners because that doesn't need to be stated, it's explicit.

The Investment Bank

In late 2004 as we wondered what was happening next, I got a timely phone call from JPMorganChase investment bank (JPMC) London to help design a new protocol. I had one white paper and benchmarks (100K pub-sub messages per second) to reach. The existing messaging layer could handle 10K messages per second, per server, and they ran a fan-out cluster of dozens of servers to reach the capacity they needed. So I wrote a prototype and demoed it, and we got the full contract.

We migrated an existing trading system off a closed message bus that was costing eight million pounds a year. I did not know how much we were saving the business... and our contracts were meager. Our design was a messaging system, and an emulation layer that let existing apps work without changes.

It was not an easy project. Hitting 10K messages per second was easy; we could do 50K in one thread quite easily. To hit 100K we had to rewrite the code to be multithreaded, and in those days it meant locks and semaphores.

It took three major redesigns of the protocol to get something we were happy with. The full history of this is on GitHub: https://github.com/imatix/openamq. The first designs were based on reverse engineering JMS. The third was based on an abstract exchange-binding-queue model (EBQ) that came to me on a beach in Lisbon, our last holiday for some time.

The nice thing about EBQ was that it defined how the server worked, formally. So your app could rely on this no matter who wrote the server. One day we met a team from RabbitMQ, who'd gotten the AMQP/0.6 spec and implemented it. It talked to OpenAMQ straight away. Nice! This is how protocol specs should be.

JPMC put together a working group to turn that spec into a "real" standard, while we continued to push our stack into production. It was a terribly hard project and taught me, as if I didn't know, the miserable nastiness of multithreaded code in C.

In 2005 my wife was pregnant and I was rushing to and from London, the bank putting us all under huge pressure to get it running. At the eight month, the baby was born, dead, and I had just a few days to grieve with her, before returning to work. I don't think we really ever recovered from that.

AMQP was not ideal. There were many problems with it, which the working group should have fixed. Instead, it descended into politics and back-stabbing. JPMC (the Chair) and Red Hat were the worst. They'd made some kind of backroom deal where Red Hat got carte blanche to rip the spec to pieces and replace it with their own version, while we (iMatix) were trying to get the rights to our code, to launch a business.

The Chair sat me down and said this: "Pieter, I want to make you a deal. If you remove your name as author of the spec, and let me tell people I wrote it, I'll get you the rights to OpenAMQ so you can start a business on it." The second part had been our agreement from the start, the first part seemed bizarre yet I was willing to make the deal, and we shook hands.

Red Hat's AMQP (so called version 0.10) is an embarrassment. No, it's an offense. It was entirely incompatible, a shoddy spec based on documenting their code, with no attempt at clarity or interoperability. And the Chair pushed this through, bullying us to accept it as a necessary compromise. Red Hat had a team of twenty people writing code and editing the specification.

Cisco eventually gave up and walked away from the project. My team gave up. We tried over and over to push AMQP towards technical improvements, proposing many RFCs for remote management, for higher-performance streaming, and so on. We did not get a single one of these into the spec.

This wasn't cheap. The AMQP working group held its grand meetings in beautiful conference rooms in London, New York, San Diego. Dozens of people attended. We filled white board after white board with TODO items. Days passed, then we all went home again, and nothing of what we'd decided happened. Meanwhile my firm paid for our own travel and hotel costs out of pocket.

My VCs eventually pulled out, saying they could not wait another six months for JPMC to assign us the copyrights. I'd made several trips to Palo Alto, & New York to get our business plans together. All into the trash.

We did go live, dealing 150K messages per second, which was excellent. The project manager told us it was one of the easier projects he'd been on. I couldn't believe him.

In retrospect, although the Chair caused huge amounts of stress and pain, he saved my firm and myself from bankruptcy. The meager contracts we got from JPMC were enough to let me rebuild iMatix as a far more interesting vehicle. The AMQP story is one that I like to complain about. Yet honestly, it was the start of something new and exciting.

Lesson: sometimes success is your greatest problem, and sometimes bad events can have great outcomes.

The Fight Against Software Patents

Around the same time, I got involved in the FFII, fighting software patents in Europe. One of my motivations was that our SMS@ application had been attacked by a patent troll (AllIsBlue). I'd fought back by building an industry association, yet was the only firm willing to take a stance. In the end I shut the app and fired that team, too.

Fighting software patents was easy at that stage. The FFII was in chaos after a long and hard fight in the European Parliament to defeat a law that would have let firms patent software, along the American model. For reasons that aren't exactly clear to me yet, I was elected president. Somewhat out of nowhere, I'd no such ambition.

Two years I spent learning all about patents and copyrights, arguing with patent lawyers and lobbyists. But by far the worse arguments were from within the FFII. It was so incredibly hard to do anything. In the end I had to create a second NGO (ESOMA), in Brussels, with its own funding, to make things work.

On the good side, I learned a **lot** and met many people. On the bad side, it cost me so much stress and money (I paid for all my considerable travel and time out of my own pocket) that I got burned out.

Lesson: don't try to fix existing organizations. Start new ones. It's sad yet there we are.

OpenAMQ, the First and the Fastest

OpenAMQ was one of the best documented and built products I've ever touched. I'll explain a little how we made it. First, though, I'll explain why we killed it.

In late 2009, the Chair and Red Hat sat down and decided, in a secret meeting, to rewrite the spec. The Chair described this in an email that has since vanished from public view, and sadly I can't show it. So you can trust me, or call me a liar. The problem they had was that the Qpid broker ran out of memory and crashed when consumers did not fetch data fast enough.

Now, this is a beginner's problem in queuing. The correct solution is to throw away messages for slow consumers, if you are working with so-called "transient" data. If you need persistent data, you have to overflow to disk and let slow consumers catch up later.

The Chair's solution was to entirely rewrite the AMQP spec. From scratch. By himself. After years and years of committee work. After years of investment by others in working code. Without asking anyone except Red Hat. And then, to force this spec through the working group using his usual tactics: bullying and lobbying.

When we first saw this new draft spec we (the sane members of the WG) were flabbergasted. There was no clear reason *why*. Nowhere on the Internet will you find a clear argument that explains why the EBQ model was broken (and I'm not defending it). Just, "here is the new draft spec, take it or go home." The only rationale we got was something like, "we're not making progress with the current spec so I decided to take over editing."

Some WG members leaped on board, impatient for a 1.0 release they could start to use. Others sat back and reached for popcorn. iMatix insisted on explanations, and we received none. It started to hit me that we were in a rigged game that we could not win. I wrote some long articles to plead for fixes to the process. No effect. We started to wind down our AMQP work and prepared to exit.

With the 1.0 release, the Red Hat vomit bucket was thrown out. We got approval from the WG to make an update to 0.9, so we did several hundred small fixes to the spec, and published that as 0.9.1. A spec for EBQ messaging, widely used, and dead on arrival. Seven years of work that took.

At some point in 2012, on a blog post about AMQP 1.0 (which is a fine protocol that entirely missed the goals of that original AMQP whitepaper), the Chair accused me of having worked against AMQP, and claimed, once again that he and his "expert team" had written the original spec. Fuck that. Yes, I had a lot of excellent input from people. Yet the original AMQP spec, every line of it, until the committee got hold of it, was my work.

I read those attacks, sitting in hospital with a sick baby, and so I pulled out my laptop and did what my Gaelic ancestors did when someone went just too far. I wrote a poem. Here it is, for your pleasure:

Dear John, you called my name? And three times in a row? _ Beware of what you call for. Well, heck, let's start this show.

You once told me, Pieter, to be rich, stop writing code, _ it's men who do the politics who pick up all the gold.

Investment banker ethics, you said when we first met, _ You proved that many times, I'll always owe a debt.

You chased hard after money, you chased power, glory, fame. _ So ten years' on we're here again, still playing at this game.

But rich friends and their stooges won't make you a good designer, _ New kitchen, car and flat TV won't make your work smell finer.

Repeat the tired old promises, your dreams of endless glory, _ because that's what they are, they're dreams, they're just a story.

You built a massive castle, and raised up a higher wall _ and invited the kings and the queens to a fancy costume ball.

And outside, we raw peasants, we toiled in the mud, _ and we built a real sprawling city, the future, my lord.

Your fortress sits quite splendid, tricked out in purest gold _ but inside those high walls it's empty, and brutal, and cold.

The sycophantic circle jerk is awesome entertainment, _ I'm breathless for the next reply, if you can maintain it.

We get it, your deep hatred, your anger, and your fear, _ these are normal emotions when your fate is crystal clear.

We're the peasant zombies, the 99% unseen, _ the dirty unwashed masses, the community, unclean.

We argue and we bicker, and we have our little wars, _ but we're the quiet storm that's breaking down your doors.

The future may remember you, John, if we care at all, _ a footnote to remind us: pride comes before a fall.

And that was the last time I heard from the Chair. Thank god.

Back to OpenAMQ

DowJones & Company asked us to replace an existing expensive data distribution system with OpenAMQ. We extended the protocol with a "direct messaging" class that we offered the working group. This reduced the message envelope size to almost zero, and batched messages that went to the same endpoint. We were able to increase performance from 150K messages/second to 600 messages/second.

This powered the DowJones Industrial Average for many years.

When we released the AMQP/0.9.1 update, it took me about three hours to make OpenAMQ work with that revised protocol. Another day for testing and updating the documentation, and we released a new version.

This was quick. Of course, we cheated, and I'll explain how and why.

When Jonathan and I started thinking about how to build AMQP we looked at abstract protocol models and decided to write AMQP as a model. This means the protocol consisted of:

- An XML file that could be compiled directly into code.
- Hand-written supporting documentation.

We used our code generator (GSL) to do the hard work. This takes models (XML files) and grinds them through code generators (scripts that turn the model into something else, like code or documentation).

Code generation has a poor reputation yet this works exceedingly well. The model is simple, high-level and explicit. Look at this XML file and you'll see what I mean. Our AMQP model has classes, methods, and fields. It lets you add rules and comments.

More, it lets you structure the model into layers. We had a lower protocol layer (ASL) that dealt with connections, security, errors, and other aspects that all protocols need to deal with. Then we built AMQP as a set of classes on top of that. Each class is a separate file, easy to edit. Trivial to add, remove, extend classes.

Push a button and you get a full written explanation of the protocol, the core of the written spec. Push another button and you get code in any language you need.

I should have seen the warning signs when I handed the AMQP/0.6 (my final text) spec to the Chair, before we'd started assembling the working group. His first act was to pull all the XML files into a single huge XML file, and try to replace our code generation tools with XSL stylesheets.

If you can't envision a complex protocol as layers, you shouldn't be in the business of protocol design.

Killing OpenAMQ wasn't as hard as you'd imagine, even after the massive effort we'd spent in building it up. And I mean massive. Look at the openamq.org site, and you'll see tool after tool. We wrote a model language PAL, for testing, so we could write a huge test set. Here's a typical test script:

```
<pal script = "amq_pal_gen">
  <session>
    <queue_declare queue = "myqueue" />
    <queue_bind queue = "myqueue" exchange = "myexchange" />
    <basic_content size = "64000" message_id = "id-0001" />
    <basic_publish exchange = "myexchange" routing_key = "myqueue" />
    <basic_get queue = "myqueue" />
    <basic_arrived>
      <echo>Message '$message_id' came back to us</echo>
    </basic_arrived>
    <empty>
        <echo>Message did not come back, this is bad!</echo>
    </empty>
  </session>
</pal>
```

This wasn't interpreted. It was compiled into code. Since our client API was in C, we generated C. It could generate any code. One test language, covering any number of client APIs. How cool is that? I wrote PAL up as an RFC and offered it to the working group. Red Hat and the Chair squashed that. I began to learn that a major problem with such groups is the ability for a few powerful individuals to keep competition away.

Anyhow, killing this product, after years of work, wasn't so hard, because:

- We didn't have a lot of paid clients, having gotten into the market too late.
- We didn't have a successful community, as this was before I'd learned how to do that properly.
- I realized AMQP was a lost cause and needed to stop bleeding money, my firm was going bankrupt.
- I was utterly burned out and wanted to stop coding.

The lessons here are numerous.

What is an open standard?

An "open standard" isn't enough to bet your business on. One of my spin-off projects was the Digital Standards Organization, and I came to understand what was needed to protect a standard from predatory hijack. I summarized the definition of a "Free and Open Standard" as "a published specification that is immune to vendor capture at all stages in its life-cycle."

What the "free" part means is, if someone hijacks your working group and starts to push the standard in hostile directions (as Red Hat did), can you fork the standard and continue? Does the license allow forking, yes or no? And secondly, does the license prohibit "dark forks," namely private versions of the standard?

If either answer is "no," then you are at the mercy of others. And when there is money on the table, or even the promise of money, the predators will move in. The AMQP experience gave me a lot of material for my later book on psychopaths.

Digistan's recommendation for standards was to use the GPLv3. We've used this in all our ZeroMQ RFCs.

The hard-earned lessons about capture also shaped my views of open source licenses, which is why today I recommend the Mozilla Public License v2 in general. It allows forks and prohibits dark forks, and is not tainted by Microsoft's long "viral" campaign against the GPLv3.

What is open source?

If we'd managed to build a thriving community around OpenAMQ, it would have survived. So the lesson here is simple: community before code. Today this is obvious to me. Eight years ago, it wasn't.

When do you give up?

In toxic projects like AMQP, when do you give up? I've tended to try to make things work until the bitter end. I'd stop only when there was no money left to invest, or literally at the edge of burnout. I've justified and rationalized this in different ways. Even now, I'll argue that bad projects (like bad relationships) are the only way to really learn.

So, a book like The Psychopath Code is mostly based on personal experience. Years of accepting abusive situations either because I did not understand (most of the time), or because I decided to tough it out. Is this worth it?

Simply walking away from a bad project can leave you damaged. It will eat at your professional confidence. You'll be afraid to try again. People will consider you a quitter (or, you'll think they do, which is more likely). Yet staying will destroy you. It'll empty your savings and leave you burned out.

The best answer I've found is the one I explain in that book. Diagnose the situation, observe carefully, intervene to turn it around, terminate when you are healed.

Please read that book, and consider how it applies to your professional life. Lessons like "keep a log." I did not enjoy the years of toxic relationships that taught me those lessons. Yet I calculate they were worth it, if the results can help others.

What's good software?

Good software is used by people to solve real problems. Good software saves people money, or makes them a profit. It can be buggy, incomplete, undocumented, slow. Yet it can also be good. You can always make good software *better* yet it's only worth doing when it's

already good.

OpenAMQ was perfect software by technical standards. It was by far the fastest AMQP broker ever. It did not crash. It had clustering and federation, remote administration, elegant logging. It was scriptable and embeddable and extensible. It was built using advanced tools that allowed one person to maintain a million lines of complex multithreaded C code.

And yet though it ran well inside JPMC, their first goal, after deploying it, was to replace it with a Java stack. They (I speak broadly) hated the tools we used. They did not understand code generation. They felt that Java could easily be as fast. They accepted the "benchmarks" that Red Hat showed them, claiming millions of messages per second, without cynicism.

And then in 2008, JPMC swallowed up Bear Stearns like a giant boa downing a crocodile. JPMC decided to switch to using Bear Stearns' rather better trading applications. The one we had ported to OpenAMQ was slated for closure. We'd received three years' maintenance, and then it was over.

While in 2010 DowJones sold their indexes division (which used OpenAMQ) to the Chicago Mercantile Exchange, and they similarly closed down the applications that were our clients.

Such shifts are common during mergers and acquisitions. They seem to have happened a lot more since 2008. In any case, by 2010 OpenAMQ was no longer "good software," and our vision of building a business on AMQP was clearly a rotten one.

And so early in 2010 we resigned from the working group. There was some fallout, some blaming. I had criticized the process publicly. The Chair got his view of history into Wikipedia and blogs, painting me as the bad guy. That was unpleasant, yet I was so tired of the arguments that I left people to interpret the situation as they wished, and focused on other projects.

Being, at that time, Wikidot.com and ZeroMQ.

Wikidot.com

In which I learn more about community. And beer. And why pizza with ketchup and mayo is a Good Thing.

ZeroMQ

In which I *finally* stop compromising my principles in exchange for the promise of money.

Before Cisco had entirely given up, we worked with them to make a high performance multicast extension to AMQP. This eventually became the first version of ZeroMQ. Though what we have today is a totally different beast.

Samsung, in Dallas and Seoul

In which I'm accused of being a cocaine addict, and we get to learn the ins and outs of Korean cooking.

The Ultimate Lessons

So much to say. I think the core lessons are: be patient, don't give up, and always be learning. You can turn even the most crappy situation into valuable lessons. Teach them to others. Be happy with what you have yet always strive to improve things. Don't let people flatter you into playing *their* games. When things get weird, keep a log. Love and respect good people. Learn to keep the assholes at a distance. Don't get hung-up on the past. Be nice to people, even those trying to hurt you. Speak up when things are bad, and tell the truth. Trust your emotions yet check where they come from. Don't be afraid of taking risks, and learn to identify and manage risks. Solve one problem at a time. Be generous. Teach others whenever you can. Remember Sturgeon's Law.

Finale

Bringing the dead machines to life was my passion for decades. Via the FFII I learned that people are the real challenge. I began to move into community building, spending a while helping Wikidot.com build their community. Yet in the end, there is nothing quite like writing some code and seeing a light turn on, and turn off again.

Thank you for reading all this. :-) If you are one of the people or firms that I talk about, and you take offense at what I wrote, sue me.

Pieter Hintjens 23 September 2016

Chapter 2. The Cretan Method

The search for truth is an ancient, and difficult pursuit. Modern society still swims in a sea of lies, in politics, business, and especially the software industry, where grand lies breed like cults. My work is to cut through the lies to develop better theories of the truth. It turns out that lies cause pain, and as we approach truth, we also become happier. In this essay I'll explain how I do this, and provide a series of tools and lessons for your benefit.

Approaching the Truth

Humanity is a powerful eusocial species, and one of our superpowers is the collection of knowledge about the real world. We do this by building models, or theories, refining them through practice, and teaching them to our young. Of all the layers of reality that we can model, the one that affects us most, every day, is the reality of other people. Karl Popper wrote: "We are social creatures to the inmost center of our being."

There is an art to shaping, testing, and applying theories. The truth is a curious thing. Like an irrational number, it exists, is not negotiable, nor subjective, yet unreachable. Truth is an absolute property of 4-dimensional space-time. We can never reach the truth, only build theories that approximate it better and better.

We construct theories apparently out of nothing. We take observations and gut feelings, and the endless legacy of theories delivered to us by past generations. We design new, or improved theories, and we encode them in language and words to be argued, remembered, and shared.

Popper argued that there are essentially two kinds of theory. There are scientific theories, which can be falsified by data or observations, and there are magical theories, which cannot. To put this another way: you cannot ever prove that a theory is true, as the truth can never be reached. You can however try, and fail, to prove a theory wrong. When you remove all the provably wrong, what is left approaches the truth.

All theories are inaccurate to some degree. However, falsifiable, scientific theories have an interesting property that magical theories lack. That is, they can be evolved towards finer and finer approximations of the truth. Here are a set of theories you will recognize:

- The ratio of a circle's circumference to its diameter, Pi, is constant.
- Pi has the value 3.
- Pi has the value 22/7.
- Pi has the value 3.141592.

These are all falsifiable, and cannot be proven true. Take the first theory. We can measure as many circles as we like, and each time we'll get approximately the same answer. We won't get any data that disproves the theory. We won't get any data that proves it. So far, so good. The second theory looks good, and is trivial to nullify. See, wrongness is scientific! The second theory is better, yet as we improve our measuring tools, we'll start to see that Pi looks more like 3.141 than 3.142. The third theory is much better, and it takes a lot of work to nullify it. There is no theory of Pi we can absolutely state is true. We only have increasingly accurate models.

Now here are some magical theories you may recognize: "C++ is great because it has a Standard Template Library," and "Java is great because it stops programmers from making bad mistakes" How can we falsify those theories? Impossible. They are not only not true, they are not even wrong, as Wolfgang Pauli would say.

A scientific theory is always wrong to some degree, and can be improved by reducing its wrongness through testing, observation, and measurement. A magical theory is untrue, and cannot be improved.

A Theory of Lies

To search for the truth, it is worth understanding the nature of deception. The Greeks thought (or pretended) that deceit was an evil spirit called Apate, who escaped when Pandora opened her infamous box. Sun Tzu wrote that "all warfare is based on deception," meaning that deception is a successful strategy in a conflict.

Most minds are honest liars. We can assume that lying is a built-in function of the human mind. The ability to lie, or at least bluff, is essential to develop theories of the world, as all theories are lies in some degree. Only a dysfunctional mind can never lie. When a mind shapes a theory, it remixes the existing theories it knows with new observations, add its approximations, guesses, assumptions, and beliefs, trying to connect those into a consistent story. Most minds do this slowly, over weeks or years. In such theories, the lies are temporary scaffolding that can be fixed over time.

Some minds, in a constant state of war with the rest of humanity, lie strategically, to confuse and manipulate. Such minds can construct theories that are entirely magical, so rapidly it happens in real time, and tell these to the listener, as utterly convincing lies. This is an assault, a weapon of war. We can call such minds "psychopathic", as their goal is to prey upon others. In their theories, the truth is temporary scaffolding, to be replaced over time by fabrications.

The value of the lie, as a weapon, is clear. In a predator-prey relationship, the predator must keep its prey confused and immobile so it can safely feed. With humans, emotional restraint is more effective and less risky than physical restraint. By injecting magical theories into the prey's mind, the predator disrupts the prey's theory process. This is one way cults trap their victims, by injecting large magical theories that disrupt logical thinking.

The first practical lesson is this: *everybody lies*, as explained by Dr. House (or rather, his script writers). I assume there are honest liars (like me, I assure you), and psychopaths. I'm skeptical to the point of mocking the entire universe unless and until it shows me data. And even then, the best it can do is "wrong", rather than "untrue".

Mind as an Evolved Strategy

Watching my 3-year old son learning, and then playing Minecraft, was a revelation. I've always treated my children as proto-adults, works in progress and yet production ready. If you've not seen Minecraft, this game is a construction simulator that is simple, deep, and social. It is the best model I've seen so far for my ideal software development process.

The old theory of "mind as clay" to be shaped by society has been nullified by twin studies. We know that our minds are largely defined at birth. "Mind as an evolved strategy that is shaped and calibrated by society" seems much more accurate. I'd like to see studies of twins raised in dramatically different cultures. Does growing up in the streets of, say, Kinshasa, calibrate your paranoia rather higher than for instance, London?

Clearly, our mental tools are sharp and functional from a very early age. From about one year old, children begin to interact with people other than their mother, and they mostly do this accurately. If you want to understand a concept like "trust", observe young children. Children start by trusting no-one except their parents and siblings. They extend trust to other adults, when their parents tell them it is OK. They trust other children implicitly, unless there is too much age difference. They do not trust unknown animals unless their parents are with them.

Similarly for a concept like "freedom", which has inspired billions of written words, and yet I've summed up as "the ability to do interesting things with other people." This is obvious from short observation of a typical child, and yet it's profoundly valuable as a working theory. I've talked much more about this in my book "Culture and Empire: Digital Revolution."

The child's world is necessarily simple, and the child's mind is uncluttered with magical theories. We are already powerful theoreticians at a young age. Children are natural scientists.

This brings us a second practical lesson: *seek childlike intuition*. The tools we need to understand the world are built-in, and evolved by long and relentless competition in the knowledge business. We are good at developing theories until our minds become poisoned by magical thinking.

Pain is Valid Data

One thing about children is how unfiltered they are. When they don't like something, they will tell you. You will get sullen faces, loud complaints, even screaming and crying. It can be frustrating to a parent. It is however fascinating, when compared to how much irritation and pain adults will accept without even blinking.

All theories are smoke, until you test them. This means, apply to reality and observe the results. A good theory works smoothly and almost silently. A poor theory creates what I'll call "friction", expressed as irritation, cost, delay, stress. For example, there are two conflicting theories in my household about how best to get to school:

1. The best way to school is by car (3 minutes' drive).
2. The best way to school is by foot (10 minutes' walk).

This morning, as we were late and felt lazy, we took the car. It took 20 minutes, most of which I spent fuming at the traffic and bad drivers. Then I realized, every moment is an experiment, and pain is valid data, and I was happy again. The car theory is nullified. We can walk again.

When you use a theory, and remember, all theories are wrong in some respects, you will always be able to feel some irritation. Often the irritation is washed out by larger concerns. It could be, "OK, I'm stuck in traffic yet at least I'm protected from the pouring rain". Or, "OK, my boss is an ass to me, yet at least I have a job."

Actual mental pain (as compared to sitting on a piece of broken glass) is a sign of serious friction, like outright lies. Minor irritation indicates small friction, like inaccurate assumptions. Every theory is either scientific, and by definition wrong, and can be improved, or it is magical and irreparable. A magical theory will cause pain when you try to use it for anything serious.

This brings us to lesson three: *improve your theories, or discard them*. The key indicator of a magical theory is that you cannot remove friction by improving the theory. You either accept the theory totally, or not at all. You can treat magical theories as a form of infectious mental disease.

The Abusive Bond

You will remark that people often stay in abusive relationships far longer than you'd expect. When we started the ZeroMQ project the core developers wanted to use C++ rather than my preference, C. They told me, "sure, it will take you ten years to learn the language, yet it's far more powerful." At the time I didn't have arguments against that. It took them five years and one real open source project to do a U-turn and discard C++.

Abusive attachment is a counter-intuitive mechanism, and worth understanding. We value relationships proportionally to our investment of time, effort, money, affection, resources. In a normal relationship this works two ways: Alexandra and Bob exchange gifts of varying subtlety, and do mental calculations to balance the books. In a "healthy" relationship, the balance is close to zero, and the calculation effort signifies the depth of the relationship. In an "unhealthy" one, there is a severe debt on one side (and the calculation effort will approach 100% of CPU, acting as a DoS attack).

There is a cheating strategy, based on future promises. If Mallory promises a large future payoff, then Bob will invest on that basis. This is the basis for many con games including the Spanish Prisoner aka Nigerian advance-fee fraud. It is the basis for most abusive relationships.

Here is how it works: Mallory makes a large promise to Bob, and offers a small gift as proof of good intentions. Bob responds with gifts, and Mallory accepts them to encourage Bob. At every point, Mallory fails to deliver, always due to tragic outside circumstances. Mallory flatters Bob and acts the victim, and asks for token investments from Bob.

In Bob's mind, the relationship is deepening and gaining value. On the ledger sits a large future promise from Mallory, and all of Bob's investments, and it feels real and deep. Bob develops an increasingly strong attachment to the theory of Mallory's future promises and instead of backing away, makes increasingly large investments.

Mallory moves from blaming third parties to blaming Bob. Everything starts to be his fault. She rewrites history to explain how he is the cause of all her troubles. Bob accepts these debts and the ledger starts to tilt massively against him. The future promise gets lost and forgotten. Bob works overtime to pay off his "debts" and to normalize the relationship, yet just enables progressively worse behavior from Mallory. The more abuse Mallory gives to Bob, the more Bob invests, and the more he invests, the more he values the relationship and is bonded to Mallory. Of course Mallory doesn't give a fig, and has a negative bond to Bob, consisting mainly of scorn and distaste.

This bond can last years, even a lifetime. Seen from the outside, it is incomprehensible and immoral. Yet it is just the price we pay for our eusocial powers: it enables a class of predators, the Mallorys. Mallory isn't always a person: it can be an organization, or a set of magical theories.

There are only two ways out of an abusive bond. One, when Bob has nothing left to offer Mallory, who will then discard Bob like trash, while explaining in great detail to everyone how it was Bob's failure. Two, when Mallory makes a demand that Bob cannot normalize. Bob may wake up at that point, or may self-destruct.

The evolved response to the Spanish Prisoner attack, in our minds, is to discount the future massively. However that is also trivial to beat: simply add enough zeros. Hence modern scams are always promise ridiculously large amounts of money or power.

Theories that are excessively complex, and promise future payoffs are a form of advance fee fraud. When it takes ten years to learn a programming language that promises you "power", you realize C++ users are in an abusive relationship with their language. C++ is the Scientology of programming languages. Java, the sprawling mass religion.

The curious thing is that when you challenge someone who is in the Spanish Prisoner's embrace, they will fight you. To question such a massive, life-threatening investment is felt as an extremely hostile act. Only when the embrace is breaking will the Bobs nod and agree that perhaps they are lost.

This brings us to lesson four: *people defend magical theories the strongest.* There are very few theories in the software world that are not fraudulent in the same way. Most software is based on magical theories, and most programmers are Bob. We almost totally lack the scientific method in software (the ZeroMQ C4.1 process is a rare attempt at this).

The Role of Emotions

Children may seem highly emotional yet look closely and you see that most children can switch their emotions on and off at will. It's been said, all children are psychopaths. (A better theory is: all psychopaths are childlike.) Children lose this ability as they develop empathy. Emotions are social communication tools, a way to manipulate others into behaving in the way that we want. They are our original, primeval language, displayed in face and body.

This is easy to demonstrate. If someone walks in your way on the sidewalk, you sidestep, smile, or nod, and the nascent irritation (the tiniest ripple, shown perhaps as a raised eyebrow) turns into a tiny pleasant interaction. However the same interaction between two cars can often lead to intense anger in both drivers. The difference is that cage that the car forms around the driver, cutting off verbal and non-verbal communication. It is easier for a car driver and a pedestrian to understand each other than two car drivers.

Lacking any response from the growing irritation, the brain flips into the "fight or flight" response you might have when someone deliberately walks in your way on the sidewalk. Road rage is a basic survival instinct caught in the wrong context. A sharp response to a

threat is safer than no response. Except of course, there is no threat.

Some people (the Mallorys I spoke of earlier) retain their childlike ability to flip emotions into adult life. They project fake emotions -- jealousy, hate, fear, anger, self-pity, sadness -- to drive others into Spanish Prisoner embraces. *Look, I'm insanely jealous! That shows I love you, now give me more affection!*

More interesting though, is our own use of emotions when we're confronted with a poorly-working theory. We will often express the pain and irritation as anger towards others. As shown by my "best way to school" example, emotion is valid as data, yet invalid as a process. Shouting at another driver is not an Aristotelian dialogue.

Emotions make great art, and terrible science. In fact we can measure the amount of magic vs. science in a process by the level of emotions. We noticed this clearly in the ZeroMQ community: as we moved from a magical to a scientific process in early 2011, all emotional arguments disappeared.

I'll explain later how to regulate your own emotions, which I call "Grounding". It is a difficult technique, yet extremely helpful in taking distance. And taking distance from your own experience is, ironically, the best way to understand it clearly. Pain is useful data only when emotions are silent.

To see friction, you can either observe pain and stress in other people, or in yourself. Observing others without responding to their emotions is already hard for non-psychopaths. And psychopaths cannot understand emotions, only read and mimic them. (I guess, without hard data, that they are terrible at science.) To observe your own experiences without engaging with your emotions is extremely hard. Yet there you are, 24/7 inside your own head. If you can develop this skill, you can see friction in practically any situation simply by engaging with it.

This brings us to lesson five: *you are your own best instrument*.

The Core Process

We've collected enough axioms and theories of theory to talk about the core process itself. If you are a regular reader of my work, you'll know this process already. Like many good theories, it's built on successful practice. The process seems simple:

- Observe friction in a social setting.
- Discover or guess the underlying theories.
- Identify the flaws in those theories.
- Improve the theories, or discard.
- Test your improved theories.

- Repeat until it gets boring.

We've seen that any scientific theory can always be improved. There is no "local maximum", only ever increasing accuracy that you can reach by spending effort and time. What is more surprising to many people (perhaps their mother told them often how special they were) is that this process is entirely mechanical. To put this another way, individual intelligence is somewhat overrated.

To value individual human intelligence is like looking at an ant colony and saying, "look, that ant is super smart!" It is magical thinking. Like ants, we're a collective species and we think in groups, not individually. The smartest ant ever to exist, working alone, is nothing compared to a few "ordinary" ants working together.

Actually, it's even worse than that. Smart people often use their intelligence to compensate for friction. The ability of humans to rationalize even the worst situations can be impressive. If there is a type of individual intelligence I value, it is the ability to sense friction and be annoyed by it. Even that depends on other people: there is no social friction in solitary.

Thus the process I just described is not quite that simple. It only works as a group exercise. I've observed a fairly typical, almost ideal cycle in our thinking process: learn from others, test the new knowledge (play), apply it to real problems (work), then teach this to others. I call this learn-play-work-teach (LPWT) for short.

LPWT has some interesting properties. First, it is innate. You will see this emerge in young children without coaching. To take Minecraft again, the process is: learn from watching YouTube videos of other kids playing, then practice alone to make sure the gained knowledge works, then play the game with other children, then teach other children the things you learned. Children learn Minecraft from other children. No books, no schools. Yet is an extraordinarily rich and deep knowledge.

Second, it works very well. This is how my open source projects build software. After many decades of refining our processes, we arrived where my kids are. We write software in tight LPWT cycles. The software is never perfect, yet it is perfectible, a scientific theory that is always wrong, yet never untrue.

LPWT is orders of magnitude more effective than classic one-way learning. It underlies a whole theory of self-organization around problems that I've addressed several times in my writing.

And LPWT barely feels like work. It is pleasurable, almost addictive. It is a curious thing to see my professional life arc back to a childish view of the world. Rather than this being a natural journey, it seems forced. Indeed, I wonder what happened to me during all those years.

Ah, yes, mass education and mass work, the curses of the industrial era along with mass media. At some point in the 19th century, the old LPWT structures broke down and were replaced by strictly separated schools, universities, and workplaces. We play as children, we learn as youngsters, we work as adults, we retire and die when elderly. Teaching is a kind of work, for a few.

Play, learn, work, die. The four phases of life. The four-lives theory has many problems apart from just making people unhappy. Let me list a few of them:

- It lets us contribute for around 45 years, 60% of our lives. Does that sound like a lot? Through the guise of a "video game", my 7-year old is teaching my 4-year old about architecture, physics, chemistry. It looks to me like LPWT lets us contribute from the age of 3 or 4 to when we die.
- It excludes large segments of the population. The high cost of education (in terms of time) favors those who have nothing better to do, which means young men. It can be very difficult, in many cultures, for women to invest in higher education.
- It favors magical theories over scientific ones. If you are going to spend several years learning knowledge, without any way to test what you are learning in real life, you are already playing Spanish Prisoner. Magical theories will be more attractive, as they can simply lie to you ("yes, learn us and you will become much wealthier!").
- It treats the elderly as waste material. The very concept of "retirement" is the sign of a hate-hate relationship between employee and employer. With LPWT, your age doesn't matter. If you can learn, play, work, and teach at four, you can do this at 84 (assuming you can hold a conversation).
- It cannot handle change and opportunity. If education is a prerequisite for work, then the already-employed cannot learn new things outside the scope of their work, except at a high cost.
- It divides the generations and ages. The working cannot teach the young. The young cannot teach each other. Instead, teaching passes through a layer called "the education system", which decides together with the political elites what knowledge it will teach. I trust you see the scope for monopolies of power and cult creation.

The 4-lives theory of society is visibly inaccurate, and generates endless friction that many of us experience for much of our lives. I think many, if not most, people stuck in this system dream of escape, a return to the freedom they felt as children. Some people do define their own lives. It is rare, and difficult, however. Society tends to frown on the itinerant and the opportunistic.

This brings us to lesson six: *society is saturated with magical theories*. That nagging voice in the back of your mind that much of your life is profoundly wrong is spot on. Modern technology and cost gravity compensate somewhat, yet life could be so much better.

LPWT is very alive, outside the formal education and employment systems. It is how we learn on-line, it is how mass web forums work. Since everyone can contribute at full speed, it is an extremely powerful collective learning approach. When people use it in anger, as Anonymous did, they become a political force and a serious threat to the established power structures.

Bedtime Stories

"Once upon a time there was a magic castle, high in the mountains..." "What color was it?" "What does it matter? OK, let's say it was white. Anyhow, in this castle lived a princess, all alone..." "All alone? How could a princess live alone? Who made the food? Who cleaned the castle?" "Look, do you want to hear the story or not?" "OK, OK, I'll shut-up."

I speak at many conferences. My style has changed over the years. I used to use slide shows, like most speakers. Neat slides highlighting key points, invoking emotions, telling my carefully constructed story. These days I come without slides, and without a script, and instead of telling my story to the audience as a monologue, I improvise a dialogue with them.

The dialogue is hard work, exhausting, yet it is easier than making slides. I've switched to that format for a simple reason. When I used monologues, perhaps a few people in a hundred would "get it". Everyone would see what I was explaining. Very few would believe me. With a dialogue, I can get half the room, or more, to get it, and react.

The dialogue is an ancient form, almost lost in modern life. We're so enamored of technology that we have forgotten this pattern. Here, watch my video. Click on my links. Friend me on Linked-in. But god forbid I ask you to negate my hypothesis, live, before an audience.

The reason I go to conferences is not to sell, or evangelize. It is to learn, as the dialogue is essentially how we do the "test" part of the core process. Here is my theory, let me explain, you tell me if you have data or observations that falsify it. If I can tell a theory to five expert audiences, and it isn't negated, then it stands. For this process to work, I absolutely have to use the dialogue.

However, and this explains the love of most speakers for their slides, if I'm trying to convince you of a magical theory, then the last thing I want is a dialogue. Instead, I invoke a different childhood scenario, the bedtime story. At bedtime, we accept any fairy tale without question.

We don't care what color the castle is, we are half asleep already, and dreaming of the high walls.

This is the power of the monologue: to send your audience into a sleep state where they will accept anything. It's well known by political speakers and preachers, by lecturers and cheap salesmen. The trouble as usual is that we have some immunity to such simple manipulation. We wake up, and then the dream is gone. The best salesmen listen carefully to their public before crafting lies that will appeal to them.

Conference slide shows cheapen the fairy tale further, by reducing it to 2-dimensional visual fast food. Here, a buzzword. There, a cute photo. Here, some memorable words! As entertainment it's fine, even elegant. As a teaching and learning tool it is tragically bad. World-class speakers with decades of experience travel to far locations, just to play text-to-voice synthesizer for 45 minutes. If you go to conferences, I bet you remember random images, yet not the stories. "The best conversations happen in the corridors." That is a statement of failure.

So we come to lesson seven: *the best pattern for learning and teaching is the dialogue*. More precisely, where one person presents a theory, where others try to negate it, and where spectators applaud and encourage. It sounds confrontational, yet if there is sufficient freedom to participate, there is no need to get it right first time, and thus no shame in being wrong. Indeed, when someone takes your work and improves it, the feeling is one of satisfaction, not embarrassment.

This is how I like to structure my software projects. It's how I organize my workshops. It's how I'd run a conference (and will, in 2015 if all goes well): multi-hour sessions where speakers present their theories (of technology) and ask the audience to negate them. There are many other things broken in the conventional conference model. Meat for another essay.

Simple is not Easy

I've heard complex designs described as "over-engineered." It's more accurate to describe complexity as "under-engineered." for it takes real work to turn complexity into simplicity. I can describe this process, as it is how I write software.

You take an existing theory, and you apply it to new problems. This generates friction, which you can resolve by extending the theory. You can do this over and over, and the theory will become wider and more complex. It is like adding more ingredients to a pizza.

If you are inured to friction, as some people appear to be, then it does not matter how large and messy the pizza gets. You'll just clear space on your mental table. However if you're easily irritated by friction, like me, untamed pizza complexity becomes a problem to solve.

You stop adding to it, and you build a new theory on top, an abstraction that has the same effect and yet is significantly simpler.

Instead of "*pizza topped with tomatoes, mozzarella, anchovy fillets, and capers*", we get "*pizza Napoletana*". Now we can mix a theory of sizes ("small", "medium", "large", "Americano") with a theory of recipes. It is easier for both customer and restaurant to use such abstractions, to test them against real pizzas ("I did not ask for onion!"), and to improve them.

Karl Popper wrote, "science may be described as the art of systematic over-simplification". Lies tend to hide in complexity, just through a process of elimination. The simpler the theory, the easier it is to negate it. If I explain a theory in ten words, that is much easier to negate than a theory of a thousand pages.

This brings us to lesson eight: *simplicity always beats complexity*.

Resolving the Gender Gap

The observation that women make better scientists than men is not new. So the gender gap in some areas, like software, is confusing to many people. I've come to believe the answer is that the mass of software engineering is simply not science. It has been this way since I started programming: the vast bulk of software written and used is based on magical theories.

Most software projects fail. This is not engineering. It is a waving of the hands over duck entrails. Oh yes, we all pretend we know what we're doing. This industry is always confident, especially when it has no clue. The truth is, we barely cling to the raft of tenability in a sea of failure.

As I explained already, magical theories are discriminatory. They take time to learn, years of study and practice, spread out over a decade or more of young adult life. Not everyone can make this investment. And not everyone is stupid enough to accept magical theories without a severe "WTF is this?" mental gag reflex. In particular, most young women are too busy building real social networks to dedicate years of their lives to the study of nonsense.

There is a reason the temples of technology are filled with young not-quite employed men. Magical theories are as attractive to young men as they are distasteful to young women. Let's compete on who knows the most arcane facts. Let's see who wins the popular vote. Let's beat each other in argument, not by using the scientific method, rather by quoting magic at each other. My cloak of invisibility beats your sword of fire!

And clerics build pyramids schemes of confusion that promise "just accept this magical theory, and teach it to others, and you too will gain power." This is how cults work. One might argue that at least technocultism absorbs the energy of that stupid-simple slice of society young men represent. However, it does seem a waste.

So we come to lesson nine: *real science welcomes everyone*, no matter their age, gender, or origins.

Intruder, Identify Thyself!

There are endless magical theories that irritate us. One that struck me was the theory of on-line identity. The Internet has become a tracker of our personal profiles. This is based on a falsehood, which I'll explain. It's a good example of how a few people can use magical theories to profit at the cost of everyone else.

In real life, identity has depth. We start as strangers in a crowd, and we expose our identity gradually, as part of establishing a relationship. I'll tell you mine if you tell me yours. To walk around with our visible name is a strange thing. We do it at conferences and weird parties. It feels forced and unnatural, because it is. Anyone walking in the street wearing a name badge is... someone to avoid.

When we start with identity and then build a social network around it, as happens on the Internet, the results are weird and irritating. It would work if we were all narcissists with an agenda to collect followers. It works for me, for instance, as I decided to make "Pieter Hintjens" my main product, some years ago.

However to operate effectively, I also maintain several other identities. Different crowds are different worlds. The identity-comes-first theory has several large problems:

- We're forced into a "do I use my real name?" choice that we rarely if ever face in real life.
- It gives us no way to divulge our identity gradually on a mutual basis.
- It forces us to use aliases for different contexts.
- It cheapens our relationships to the point where "friend" has lost its meaning.
- It allows centralized brokers to effectively own and tax our relationships.

These problems go beyond irritation into full negation of the theory. The term "Facebook divorce" gets 145M hits on Google.

As with other magical theories, the flattening of identity is also discriminatory. It promotes the trolls and attention-seekers and obscures the modest, the calm, the reasonable. It is the reason we get such inflamed and wasteful arguments about gender and race and politics on

our forums. Pseudonyms make no difference. A karma whore would smell as bad, by any other name.

So we come to lesson ten: *the core concepts of our Web are magical theories*, which cannot be fixed.

Conclusions

This essay is an explanation of a personal approach to learning that I call, lacking much imagination, the Cretan Method. Everything I've written is wrong.

I've explained how society collects and transmits knowledge of the world in the form of theories. Some theories are wrong, and the rest are entirely untrue. We can always improve a wrong theory (also called a scientific theory) by testing it, feeling the friction its wrongness causes, and fixing the theory accordingly. We cannot improve untrue theories (also called magical theories).

We're born as scientists. Society bends most of us into believers of magic, and if we're lucky we escape that, and return to being scientists. Large segments of society are still based on magical theories. Over time, these will die and be replaced by increasingly accurate scientific theories. One example is the learn/work/play trichotomy of modern society. The intuitive, and much more effective approach is to mix learning, working, playing, and teaching into a smooth, continuous cycle.

I've suggested that the best way to test theories of social behavior is to use yourself as an instrument. This means being ready to measure your own experience without emotional responses. I'll explain the technique of regulating your emotions -- Grounding -- in the last section.

And I've suggested that while a monologue (the sales pitch or slide show) is a good way to sell magical theories, the best way to teach a scientific theory is the dialogue, where students search for points of friction with a theory, or even negate it if possible. A free and open dialogue can rapidly negate untrue theories, improve wrong theories, and generate entirely new theoretical abstractions. My belief is that all knowledge work would be best structured as a dialogue.

Finally, we come to a process that could be described as "pain driven development". Take a theory, test it against the real world, observe your own or others' pain, and use that to guide improvements in the theory. It is a mechanical social process that needs no special intelligence. If you really want to flatter the individual, perhaps some people are most sensitive to friction and faster to develop new theoretical models. And others are better at ignoring friction and surviving in a magical landscape.

Grounding

Grounding is to remove all negative emotions from your mind, leaving happiness and clarity. It is one of the most valuable things I've learned in my life. Over time I learned how to ground myself consciously, and rapidly. Unlike meditation, which requires peace and calm, grounding works even when people are deliberately provoking you.

Grounding has echoes in Cognitive Behavior Therapy, a technique developed by therapists who engage with the emotionally extreme minds described as "borderline". I suspect CBT is more useful to therapists, than to their patients. Grounding is simpler, and is meant for self-application in the living room and workplace.

The goal is to remain calm and happy no matter how difficult things become, and in that state, to accurately observe yourself and others. Only with accurate observations can you improve and negate your theories. Emotional turmoil can easily become a cage. So Grounding is a path to freedom and opportunity. It lets you experience the very worst situations as "useful data" rather than "destructive pain".

Let me first describe the Grounding process, and then provide more detail. We start by identifying our strongest negative emotion, and naming it. We then reverse engineer it, using the theory that all emotions are a form of social communication. We then find the root theory that the emotion is based off. We then negate that theory, in our own minds. The emotion then goes away.

This is not easy. Expect years of reading these words and not getting it. Some emotions are trivial to debug and fix. Some are extremely difficult. It all depends on the power of the underlying theories, which can take years to construct and years to dismantle.

Let me do a "Hello, World" walk-through of an emotional moment. Alex is on a date with Brita, and Brita says she has to go home soon. Alex is suddenly filled with self-pity and pain. He feels Brita is deliberately hurting him. He feels angry at her, hides it, and then when she says, "because my cat is sick and I want to check on him", feels guilty about having such a strong response to her words.

Classic Alex. Now let's debug this little story. The guilt comes from fear that she'll have seen his anger and react to it. The anger is a "fight or flight" response to the pain he feels. The pain comes from fear of rejection, and being alone. The self-pity is a cry for help, the emotion a baby feels when it cries for its mother.

Clearly Alex, who is mostly thinking about the possibility of sex with Brita, isn't going to be regulating his emotions much, if at all. Luckily, Brita was lying about her cat (in fact she's married and her husband is waiting for her) and she's also being flooded by doubt, guilt, and fear. She's not really reading Alex at all.

Many years later, a now wiser and less reproductively stressed Alex meets Brita again. They chat and have a coffee. Later, Brita once again says she has to leave. Alex feels the smallest kick of self-pity and conducts a small internal dialogue. "OK, self-pity, what's your theory?" he asks. Self-pity answers, "if you cry, she might give you a hug. Then, sex!" Alex mocks self-pity's theory. "Did that ever work?" Self-pity grudgingly shakes its head and goes away.

That is the general approach. Let me now list the main negative emotions and their classic patterns:

- **Self-pity**: a cry for adult affection. Self-pity will rapidly turn to anger as our imaginary adult fails to comfort us. The theory is, "I'm under seven years old." This is the easiest emotion to name and debunk.
- **Jealousy**: a cry for attention in competition with another person. Jealousy will also turn to anger as our target fails to respond. The theory here is, "I can bully this person into spending more time with me". It is harder to debunk, as it is often true for a while. However, a good negation could be, "it's not attractive to act like a child."
- **Anger**: showing your determination to fight. The underlying theory is, "showing my anger will bend the other person to my will." You can break this theory in various ways depending on the context. For example, "He did it by accident", or "there is absolutely no hope of saving this relationship", or "he enjoys seeing me angry", or "I can resolve this more profitably without risking a physical violence".
- **Fear**: showing your stress at a looming danger, such as being rejected and alone, or being hurt. The theory is, "if I show fear, mommy or daddy will come and save me." The simplest negation is: "no they won't, you have to survive this alone." A more subtle one is, "the danger you feel is false because X and Y," which applies to most social fears.
- **Sadness**: showing your stress at losing something precious. The underlying theory is, as with fear, "mommy or daddy will come and make things right again." We can debunk that theory quite easily in most cases. "Will being sad get X back? What are the chances, honestly?"
- **Shame**: showing our stress at being diminished before other people. The underlying theory is, "we will be ostracized, and then die alone and starving on a rocky hillside." It is easy to debunk, with: "frankly, no-one cares enough to ostracize you", though that will lead to self-pity. Perhaps to be more diplomatic, "everyone is so busy with their own problems that no-one will see your pants are down."
- **Guilt**: showing our fear of being shamed. The underlying theory is, "if we show how bad we feel about doing bad things, maybe people will forgive us." This is surprisingly easy to debunk, with, "everyone makes mistakes, you just apologize sincerely and carry on",

and sometimes, "you didn't actually do anything wrong, the other person is manipulating you."

- **Loneliness**: showing our hunger to belong in a group or family. The underlying theory is, "people will want us because we look sad." And the negation is, "that works for babies only. For adults, a smile takes you a lot further than a frown."

It may sound strange to conduct an internal dialogue with your emotions. Yet this is the Grounding technique: debunking the emotion by explicitly stating, to yourself, observations and data that negate the underlying theory that the emotion is working on.

When you start, use pen and paper. Your mind won't be able to run the internal dialogue properly. So you search this list for your strongest emotion, and you write that down, and you then ask yourself "where does that come from?" Emotions often trigger other emotions, so you may have to unravel several layers to get at a root theory. When you do this more and more often, you find yourself Grounding yourself on-the-fly, so the layers don't build up.

It can also be helpful to practice this with other people, if they trust you enough to go there. "What are you feeling now?" can be an interesting starting point. If you ever find yourself acting as therapist for an emotionally abusive person, these techniques can save you from the destructive burnout that comes from engaging with unrepentant emotional storms.

More broadly, being grounded lets you meet and talk to anyone without engaging in the emotional games most of us play subconsciously. This includes psychopaths, who attack the emotional surface area of their targets. When grounded, you offer no opportunity for manipulation, and you become a "grey rock" that psychopaths will gloss over.

Thanks

This essay has been many years in the making, and is the result of endless dialogues with friends, and strangers who rapidly became friends. I'd like to thank everyone who's helped me figure these things out, particularly Sachiko, Tuuli, and my mother.

Chapter 3. A Hundred Tiny Slices

In which I wrote a "Top ten..." Buzzfeed-like article, and people seemed to like it, and asked for more. I stopped at the tenth article. Here they are.

Ten Tips for Young Programmers

Writing code is like doing magic. Just say the right words, and amazing stuff happens. Or, horrible things happen. Becoming a great programmer isn't easy. It takes time. For young coders, I'm going to give ten tips to help you along that path.

1. Take Your Time

It takes years to get good at coding, so be patient. Your first projects will be weak, no matter how smart you are. If you're coding every day you'll be decent after five years, and good after ten. And after twenty years you may become great.

1. Write Code, Write Code

There's a myth that superb coders are born. Yes, you need talent, yet above all you need practice. Code every spare minute. You need to practice for years, alone and with others. It won't make you rich, it will make you better. Coding is like playing music. Play the same tune a hundred times, and each time you will learn something new.

1. Develop Your Strengths

Discover what you are good at: everyone is different. Maybe it's problem solving. Maybe it's teaching others. Maybe it's long, deep focus on difficult concepts. The best sign you're good at something is you enjoy doing it. Over time you'll become better at other aspects of coding.

1. Learn to Work With Others

Alone, you'll always be mediocre. Only when you work with others can you really shine. Learn to let others compensate for your weaknesses. Look for teams that need you and that can challenge you. Join open source projects. Learn open source culture. Learn from others, and especially their mistakes.

1. Use Science, not Magic

Most of the software industry is plain wrong. Learn to recognize and reject magical arguments. Science is about solving real problems with wild, irresponsible answers, and harsh error correction. Don't follow fashion. Take the next most urgent problem. Write a

minimal plausible solution. Test that, and keep it only if it seems to work.

1. Trust Your Instincts

We're born with good instincts for working with others. Education and work beat these out of us. Most of all we learn to accept discomfort and pain. Learn to trust your instincts. If something you're doing seems wrong, fix it. Even if it takes a decade: write it down, understand it, and fix it.

1. Work With What You Have

Work with the problems, tools, and people you have at hand. Focus on getting it right, minimal, and modest. Don't wait for tomorrow's technology to arrive. Don't try to invent the future. Just get to work making real solutions to real problems. The future will invent itself.

1. Embrace Criticism

Lose your ego. When someone criticizes your work it can hurt. Yet it's far worse to be ignored. Ask people for critiques. Make your work public and open source, when you can. Now and then you'll hit a troll who really tries to hurt you. It's never personal. Learn to shrug that off.

1. Keep Your Costs Low

Cheap is important. Learn to use Linux, and a cheap or second-hand PC. Learn the command line. Stick with small languages like C, instead of massive languages like C++. Learning a larger language does not make you a better programmer.

1. Publish Your Work

Put your code out there, using your real name. Become a contributor to open source projects. If they don't want you, find projects that do. Build up your public profile, e.g. on GitHub. It's your future resume.

Ten Habits of a Good Programmer

Good programmers are born and then made. You need talent and passion, and you need experience. I was lucky to learn from the best, and have lots of opportunity to practice. I'm going to summarize ten of the top habits I've always applied to my work. Cause or correlation? You tell me.

1. If it works and is still useful, don't throw it out

This means writing portable code, and making reusable pieces. Libraries, APIs, whatever it takes. Be your own client. Layer your work.

1. Never solve the same problem twice in parallel

This means making tools. Adopt the Unix philosophy: command-line tools that do one thing and do it well. Learn about pipes. While you're at it, use Linux (or a BSD) as your development box.

1. Solve the same problem often in serial

This means being willing to throw out your code and rewrite it when you find better solutions. If you work towards minimal APIs rather than features, this works better.

1. Write code, and repeat, until you are fluent in your language

Using a small language makes this easier. It takes years to become really fluent in a programming language.

1. Learn to use code generators

I've spoken of GSL before. A general-purpose code generator is an essential tool for a serious programmer. There are a few options. Try them, choose one.

1. Work with others

Learn the techniques of collaboration. A good way is to contribute to an open source project. Or even better, start your own. Read Chapter 6 of the ZeroMQ Guide. It will enlighten.

1. Technology is a tool, not a tribal affiliation

Never turn pragmatic choice of tools into a belief system. I have written editors and code generators in COBOL 74. The language matters little.

1. Aim for this cycle: learn, play, work, teach

It is the fastest way to get better and deliver code that others can use and trust. Avoid this cycle: imagine, argue, agree, work. It is the fastest way to deliver junk.

1. Get your edit-compile-run-fail cycles down to seconds

If your language offers a REPL that's cool. If not, make a shell script that runs the loop for you. Your goal is that after every edit, your code self-tests and passes or fails. Use asserts or equivalent to challenge your own code.

1. If you need debuggers, you're doing it wrong

Make one change, test one change. Layer your code and write self-tests so that when it works, it always works. Use print statements. Aim towards APIs that are fully testable. Avoid language patterns that deliver fuzzy, unclear internal APIs.

Ten Rules for Good Code

It's one of my interview questions: "what is Good Code?" Surprisingly, almost no-one gets it right. It's not about speed, elegance, language, or style. *Good Code is code that solves real problems for real people, in an effective way.* Let me list the top 10 rules for writing good code.

1. Use Git and Github

I'm not going to dignify this with a non-zero number. If you aren't using git and github.com then you are already making excuses for doing it wrong. It is not about fashion or groupthink. Github (the company) know how to make Good Code, and their tools reflect this. Update: GitHub seems to be losing the plot, so this advice may get dated. Consider GitLab.

1. Use Problem-driven Development

Identify the next single smallest problem you want to fix. Write the simplest plausible solution, and test it. Your commit message states the problem, and then your solution. If there is a single change you make to how you code, please, this one. Don't make features. Solve problems.

1. Make it Open Source

Not eventually. Not when it's working. Not if you get management approval. Now, today, before it's finished. Before you even decide what you're making. Before your first commit, even. Open source means people, and that means voices to guide you to solving the *right* problems.

1. Always Be Making APIs

Write every piece of code as the body of an API. Read "Ten Rules for Good API Design". Be generous with new repositories. Layer your projects like you're making lasagna. Document your APIs, use them, publish them.

1. Don't Document the Code

Documenting code is like writing "Tasty!" on the side of a coffee cup. If the code isn't readable on a grey Monday morning before coffee, chuck it out and start again. What you document are APIs, so that can mean methods and classes. That is fine. Explaining what this funky loop does is not fine.

1. Style for Readability

You do need a style guide. And it must be strict. All that matters is readability. Collect patterns that work in your language. Document them, improve them, and make sure the *entire codebase conforms*. Do not use automatic reformatters, as they hide mischief. A good

style is worth applying by hand.

1. Better is the Enemy of the Good

We already know this yet it's worth repeating. Do not optimize your code. Aim for "OK" and then stop. If you are following rule #1 then slow performance will pop up as a juicy problem somewhere down the line. It will always, *always*, be in a place you never thought of.

1. Use Your Own Work

It's shocking how many programmers don't use their own work. Then they have the arrogance to lecture users about "what they really need." Look, if I brew beer and make pizza, then I'm damn well going to sit at my table. Good Code is so delicious you can't resist it.

1. Use Code Generators Carefully

Code generation is powerful stuff. I'm speaking of industrial-scale tools like GSL. If you don't use it, you're handicapping yourself. If you over-use it, you get insanity. Learn to look for the sweet spot where it's simple and obvious.

1. Make Portable Code

When you have to get into system internals, hide these behind abstract APIs. Your code should run on every box there is. If it doesn't, then invest in portability frameworks. This always pays off. Portable code is always simpler and cleaner than the weird stuff that thinks it's special because it does voodoo.

1. Lie to Your Management

Maybe you have an enlightened management that knows what "open allocation" means. Or maybe they're going to be one of your main problems. When they demand schedules, deadlines, designs, and architectures, lie to them. If you cannot do this, find someone who can. It is easier to get forgiveness for success than permission.

Ten Rules for Good API Design

Every software developer uses APIs and most of us make them. The design of a "good" API is a black art. You know one when you see one. And yet how many of us could explain why some APIs are complex and hard to learn, while others are clean, simple, and a joy to use. It's a question I'll answer in this article, and provide ten rules for good API design.

1. Make Only What You Need Today

This is the top rule. Only solve problems you must solve, and make minimal answers to them. The temptation to solve tomorrow's problems is huge. Resist! Instead of trying to ship code in advance, focus on reducing your shipping cycles. If it takes you a few hours to ship answers to new questions, you can stop guessing what tomorrow's questions will be.

1. Make the API modular

Divide large problems into smaller ones, and solve each one separately. A modular API is easier to learn, and change over time. You can deprecate old modules with new ones. You can teach modules one by one. You can separate experimental pieces of the API from stable ones, from legacy pieces.

1. Use Structured Syntax

Use a structured syntax for the API: `thing.action` or `thing.property` instead of `do_action_with_thing`. The syntax naturally fits a modular approach, where each module is a class of things.

1. Use Natural Semantics

Invent no new concepts. Use only concepts the developer will already know, as the basis for your class system. If you find yourself having to explain a concept, you are doing it wrong. Either you're solving future problems, or you're structuring the API wrongly.

1. Make the API Self-consistent

Be rigorous in using the same style and conventions in every class. Consistency means when someone learns one class, they've learned them all. Document your conventions and make them a required standard for contributors.

1. Make the API Extensible

Easy extensibility has many benefits, not least it welcomes contributors. It also lets you delay implementing features, with "it's easy to add later if we need it." Every feature you do not add today is a win.

1. Make it Fully Testable

Every class and method must be fully testable by hostile code. Write your tests as you write code, and use the tests as documentation of the contracts that the API provides to the outside world. Run these tests every time the code changes. Do not worry about code coverage. What matters is the external contract. Consider using contract lifecycles.

1. Grow by Layering

Keep your APIs focused and grow them over time by layering new APIs on top. Extensibility does not mean indefinite growth. Be explicit about the scope of the API, and enforce that scope.

1. Keep it Simple to Use

The ultimate test is how simple the API is to use. When you write an example, could your code look simpler? Are you forcing the user to specify options they don't care about? Are they taking extra steps that add no value? Be obsessive about reducing the visible surface area of your API.

1. Keep it Portable

Do not allow system concepts to leak into the API. Abstract them cleanly, with the intention: this API could run on any operating system. Your API must hide the implementation, though be careful about rule #4, and use natural abstractions.

Ten Laws of the Physics of People

When we make software, laws of physics apply. I call this the "physics of people." When we ignore these laws, our work collapses like a badly designed bridge. Maybe you didn't realize Einstein's Equivalency Principle applied to you? Then read on, and be enlightened...

1. Newton's First Law of Motion

Everything we do has an economic motive. Things only move as a result of energy coming from somewhere else. People and organizations also follow this law. Economic motive is a force. The more accurate the economic motive, the more accurate the direction of movement. If a factory produces 500,000 rubber boots because the CEO has a boot fetish, that's inaccurate. If it's responding to rising sea levels and demand, that's accurate.

Lesson: give every team and individual access to accurate economic incentives.

1. Newton's Second Law of Motion

The bigger the team, the more force it needs. Sometimes it's a bike. Sometimes it's a car. The smaller the thing you want to move, the easier and faster it moves. If you try to change a 5,000 person company, you need massive force. To change a two person team takes a few beers and a single client.

Lesson: smaller teams move faster than larger teams, given the same force.

1. Newton's Third Law of Motion

When you push an organization, it pushes back. Or, put this another way, if you try to change a 5,000 person company, you will be fired. Unless you're the CEO. Then you'll be given a huge bonus, and then fired.

Lesson: build fresh organizations instead of changing existing ones.

1. The Equivalency Principle

Falling is indistinguishable from making progress. Until your face hits the pavement, you will believe you're in control. Since all movement is a reaction to an external force, you can only judge your success by aiming for the pavement, and failing.

Lesson: the bigger your leaps, the more damaging your failures.

1. Murphy's Law

If it can break, it will break. In fact the full law adds, "*in the worst possible way, at the worst possible time.*" Failure isn't just all around us, it's inevitable. We cannot make perfect systems. Rather than trying to prevent failure, we need to learn to embrace failure, and make it part of our process.

Lesson: embrace failure as the best way to learn what works.

1. The Uncertainty Principle

The more you know about one topic, the stupider you become. Or, as my mom used to tell me, "never trust someone who has all the answers, especially yourself." Experts are dangerous, if they are not balanced by naive laymen. Diversity is more valuable than expertise.

Lesson: diversity is not a political slogan. It's the basis for collective intelligence.

1. Zipf's Law of Power Distributions

20% of any system always has 80% of the power. It applies to cities, languages, earthquakes, and economies. And organizations, and software systems. You'll spend most of your effort on a fraction of the software. Over-engineering code that isn't in the critical path is a waste of time.

Lesson: if shitty code solves the problem, it's not shitty code.

1. Moore's Law of Cost Gravity

Cool stuff gets 50% cheaper every 18-24 months. Whether we like it or not, our software lives on an exploding cloud of silicon. Pavel Baron said, "architecture is dead." Even a single program behaves differently than its authors believe. This is why we need profilers, to optimize code. A system with many pieces is unknowable.

Lesson: the size of our systems -- the number of moving pieces -- doubles every 18-24 months, and *no-one fully understands any system.*

1. Amdahl's Law

The more you need consensus, the less work you can do. If you spend one hour a day in meetings, that limits your effective team size to eight people. *At best.* In practice most of us spend a lot more time getting consensus before we can start working. Do you need approval to start on work? Do your team have standards for language and platforms? Do you have to wait for approval to put your patch into production? These are all wait states. The maximum effective size of most teams is probably less than five.

Lesson: read about the Actor model, and become a message-driven, zero shared state Actor.

1. Conway's Law

The software you make looks like your organization. If you are in a shitty organization, you will make shitty software. To grow a large scale decentralized system (and nothing else survives these days) you grow a large scale decentralized organization. If your organization causes you pain, the software it makes will cause pain to its users.

Lesson: if you're not happy in your job, build yourself a better job.

Ten Rules for Open Source Success

Everyone wants it, lots of people try it, yet doing it is mostly painful and irritating. I'm speaking about free software aka open source. Today I'm going to summarize 30 years of coding experience in ten management-proof bullet points.

1. People Before Code

This is the Golden Rule, taught to me by Isabel Drost-Fromm. Build community, not software. Without community your code will solve the wrong problems. It will be abandoned, ignored, and will die. Collect people and give them space to work together. Give them good challenges. Stop writing code yourself.

1. Use a Share-Alike License

Share-alike is the seat belt of open source. You can boast about how you don't need it, until you have a bad accident. Then you will either find your face smeared on the wall, or have light bruising. Don't become a smear. Use share-alike. If GPL/LGPL is too political for you, use MPLv2.

1. Use an Zero-Consensus Process

Seeking upfront consensus is like waiting for the ideal life partner. It is kind of crazy. Github killed upfront consensus with their fork/pull-request flow, so you've no excuse in 2015. Accept patches like Wikipedia accepts edits. Merge first, fix later, and discuss out of band. Do all work on master. Don't make people wait. You'll get consensus, after the fact.

1. Problem, then Solution

Educate yourself and others to focus on problems not features. Every patch must be a minimal solution to a solid problem. Embrace experiments and wild ideas. Help people to not blow up the lab. Collect good solutions and throw away the bad ones. Embrace failure, at all levels. It is a necessary part of the learning process.

1. Contracts Before Internals

Be aggressive about documenting contracts (APIs and protocols) and testing them. Use CI testing on all public contracts. Code coverage is irrelevant. Code documentation is irrelevant. All that matters is what contracts the code implements, and how well it does that.

1. Promote From Within

Promote contributors to maintainers, and maintainers to owners. Do this smoothly, easily, and without fear. Keep final authority to ban bad actors. Encourage people to start their own projects, especially to build on, or compete, with existing projects. Remove power from people who are not earning it on a daily basis.

1. Write Down the Rules

As you develop your rules, write them down so people can learn them. Actually, don't even bother. Just use the C4 rules we already designed for ZeroMQ, and simplify them if you want to.

1. Enforce the Rules Fairly

Use your power to enforce rules, not bully others into your "vision" of the project's direction. Above all, obey the rules yourself. Nothing is worse than a clique of maintainers who act special while blocking patches because "they don't like them." OK, that's exaggeration. Many things are much worse. Still, the clique thing will harm a project.

1. Aim For the Cloud

Aim for a cloud of small, independent, self-organizing, competing projects. Be intolerant of large projects. By "large" I mean a project that has more than 2-3 core minds working on it. Don't use fancy dependencies like submodules. Let people pick and choose the projects they want to put together. It's economics 101.

1. Be Happy and Pleasant

Maybe you noticed that "be innovative" isn't anywhere on my list. It's probably point 11 or 12. Anyhow, cultivate a positive and pleasant mood in your community. There are no stupid questions. There are no stupid people. There are a few bad actors, who mostly stay away when the rules are clear. And everyone else is valuable and welcome like a guest who has traveled far to see us.

Ten Steps Towards Happiness

Today, I'm going to explain ten simple steps you can take towards being happier, and making those around you happier. It takes no money and no magic. Just a shift in how you see the world, yourself, and others. As the 14th Dalai Lama said, "Happiness... comes from your own actions."

1. Invest in your senses

Your body and mind need a good diet. Make your own food, and make it tasty. Take photographs you find beautiful. Play music that excites you. Wear clothes that feel nice. Walk through the city at night, in the rain. Feel the texture of your life, with all your senses.

1. Do things you are bad at

Learning makes us feel alive. Challenge yourself, and keep proving you can learn. Learn to juggle, to hold your breath underwater for longer, to solve a Rubik's cube. Learn to play music and play for yourself. Learn to paint and draw.

1. Enjoy other people

Nothing makes us happier than other people. Take the time to talk to people you cross paths with. Be nice, chat, smile. If you live far from other people, move to be closer. If you drive a lot, change your life so you can walk and cycle. Go out and mingle.

1. Be part of bigger things

Being part of bigger things makes us feel valued. Find communities and projects to be part of. Start your own. It could be on-line or in the real world. Even the smallest contributions can make a difference to someone. Be precious to strangers.

1. Finish your projects

A sense of achievement gives us hope and confidence. Write down your life's accomplishments. Even small ones. Now keep this list up to date. Every year, think of what you've achieved. Finish your projects and move on to new ones.

1. Get rid of bad actors

Nothing can make us more miserable than other people. Read "The Psychopath Code," and learn to recognize bad actors. When someone is a source of misery, remove them from your life. Whether it takes a day or a year, make this a project, and finish it.

1. Ground your emotions

To be happy you must deal with negative emotions. Learn to recognize these in yourself, and deal with them. Anger, self-pity, jealousy, fear, hate, loneliness... set them aside, and let happiness take their place. If you drink to relax, cut it out.

1. Revalue your time

Stop wasting your time on commuting, boring jobs, meetings, TV. Do only things that you feel are worthwhile, with people you like. If this means a cut in income, so be it. Be the person you really want to be. Don't take it all too seriously, we all die.

1. Travel light

Material possessions are more often a burden than a pleasure. Give away or sell anything you own that does not make you happy. Do not let your possessions define you. We live in a world of plenty. That means you can own less, not more.

1. Want nothing, accept everything

Above all, explore the world without desire or demand, and be tolerant of whatever happens. Most people are nice, and even the others teach us. When you want nothing, you cannot be disappointed. When you accept everything, you will see beauty in every moment.

Ten Signs of a Psychopath

One in 25 people is a (sub-clinical) psychopath. You may meet them at work, on dating sites, in the street. If psychopaths have one goal, it is to take from others and give nothing back. If they have one supreme talent, it is not getting caught. Everyone likes a psychopath. Until it gets personal, and then your life starts to bend. I'll list ten ways a psychopath (let's call him or her "Mallory") touches your life. If you recognize all of these, beware...

1. He's really into you

Mallory likes you and wants to hang out with you. A lot. He thinks you're special, interesting. You wonder why he's doing this. Did you suddenly become a lot more attractive? Yet it's flattering and makes you feel good about yourself. So you don't ask if he's too good to be true.

1. He likes himself... a lot

Mallory dresses well, works out, enjoys showing himself off. He spends a lot of time at the gym. He has nice sunglasses. He cares a lot about how he looks, and this is constant, no matter the occasion. He takes a lot of selfies and posts them online. You feel like he's talking to people you don't know.

1. He leaves deep impressions

It goes further than just fitting in. Mallory leaves a lasting impression in people. You will notice that people talk about him like he's a celebrity. He has some way of projecting his character that people adore. It makes you vaguely jealous.

1. He is rude to unimportant people

While he charms to your friends and relations, he is dry or even rude to people who are beneath him. For example in a restaurant he will not thank the staff who serve him. Unless he sees someone he likes, and then he is charming. You feel uneasy with his behavior.

1. He asks you for small favors

I call this "greasing," and it's a thing Mallory does to make you like him. He asks you for small innocent favors. Helping him move stuff. Advising him with something. Just keep him company. Every time you say "yes" you become easier to manipulate. You start to like him. It's called the Ben Franklin Effect.

1. He gets angry easily

As you spend time with Mallory you'll have small, stupid arguments. He'll misunderstand something you said, and be angry with you. It will blow over, after *you* apologize. He never apologizes. This is another classic tactic to rope you into a "relationship" with him. You begin to feel anxious when you talk with him.

1. He has no history

He has no old friends you can speak to. He has lots of stories, yet nothing you can check on. If you google him, there is little except his self portraits. He has no professional track record. His past is blank and his present is unstable. This makes you curious. Yet he's open to everything. Think of the possibilities.

1. He is super social

He is fast to make friends with your family, colleagues, anyone who matters in your life. He does this with energy. Pretty soon everyone has accepted him and likes him. If you have any doubts about him, others will tell you he's OK. This soothes you.

1. He needs things from you

He doesn't hide that you can help him. He may even say, "only you can save me." It is never about money. Maybe he wants to save the world, and you can help that happen. Over time, his needs become your goals. He makes you feel like the most important person in the world.

1. You start to change your life

You begin to shift your priorities to suit Mallory. You begin to break old habits, and spend less time with people you usually hang out with. You feel energized and positive. Mallory promises you so many things!

Does this describe someone in your life right now? Do you want to know what Mallory is really thinking, and planning? Do you want to learn how to break it off and get away? For more on psychopaths, read "The Psychopath Code." It's available on Kindle, in paperback, and as a free PDF.

Ten Myths about Harassment

Today I watched a revealing video of Yale students with a professor. The mob insult and harangue someone with decades of experience defending free speech. It goes on far too long and leaves us disturbed. These young people act like a pampered, idiot mob. And yet you cannot deny their deep anger. Who is the harasser, and who the harassed, in this video?

Comments are welcome

I'm going to propose one answer at the end of article. First, a typical definition: *"Harassment is any form of unwanted and unwelcome behaviour which may range from mildly unpleasant remarks to physical violence."* You should immediately see a problem: it is broad and subjective (violence apart). Next, let me suggest ten myths about harassment. These are claims or assumptions people often make when discussing harassment.

The Myths

1. Harassment equals discrimination

Discrimination is prejudgment of individuals based on recognizable yet irrelevant criteria: ethnic origin, gender, religion, appearance, etc. It is often accompanied with harassment, used to implement discriminatory policies. Yet these are not the same. Much, even most harassment has no element of discrimination.

1. Harassment is a gender/race issue

This is the same myth, restated in terms of the feminist and "minority" struggles against discrimination. The danger with this myth is that it misses the bulk of harassment, misunderstands it, and ironically, makes it harder to fix. Worse, the myth splits us. We solve harassment together, not divided.

1. Harassment is rare so it's irrelevant

In fact slow, methodical harassment is widespread. It is often covert and indirect: violence against our time, space or belongings. Lies and distortion of events. Attacks on our reputation. Neglect of us and our environment. Intrusions into our privacy. We are so blunted by this that we shrug and dismiss harassment as inevitable.

1. Harassment is an inevitable part of life

While harassment is common, it is not inevitable. Many groups are free of it. Yet others are riven by it. I've seen both cases in my work with communities. It shows in people being broadly happy, or broadly miserable. Consider it a form of infectious disease that can hit any group, where the goal is 100% eradication. It takes science, knowledge, and patience to cure a disease.

1. Harassment is obvious when you see it

Since we've learned to unsee harassment, we assume it must be dramatically visible when it happens. In fact, most harassment is a chain of small, almost invisible acts spread out over time. The drama comes at times, yet it hides the far larger burden of costs. Looking for drama is counter-productive. We must instead look for damage.

1. Anyone can be the harasser

Like the best myths this is half-true. We're all capable of joining a mob, as in that video. Temporary power over others can be addictive: see how predatory that mob is. Yet most harassment is a long term, focused activity. Anyone with empathy is jolted when they realize they are hurting someone else. It can take a while for empathy to react. We can all make mistakes, yet most of us self-correct.

1. Harassment is a motiveless act

Harassment is driven by a hunger for power, unencumbered by empathy for others. It is the tool of a small slice of people who use it to capture victims, attack rivals, discredit critics, and divide opposition. Such people learn the techniques young, and practice them all their lives. They are almost invisible, except by the damage they do to others.

1. Adults will police each other

Sometimes this works, yet it is not reliable. People tend to accept the word of the most dominant, charismatic individual in a group or couple. Charming lies survive for years. Did you learn the tongue taste map at school? It is 100% bogus. If the harasser is assertive, they can turn a mob into their tool. Most of us are too polite, afraid, and timid to police others.

1. Outlawing harassment will stop it

There are those who follow rules and social mores. And there are those who disrespect others, and harass them when it suits. Bullying is banned in every civilized school, yet is widespread. Rules without fair and accurate enforcement is worse than no rules. Harassers are the first to exploit poor rules and weak authority.

1. We don't need rules, we have manners

Look at anti-harassment rules differently. They are not to tell the harasser what not to do. Rather, they are to tell the rest of us when to ask for help. Weak rules say, "Don't do A or B." Good rules say, "X and Y are not acceptable. If this happens to you, follow procedure Z."

Conclusion

I'm a reluctant authoritarian, assuming everyone has the freedom to walk away from corrupt authorities. I've seen groups riven by harassment (like the FFII I was president of for two years), and groups almost entirely free of it (like the ZeroMQ community). The difference is not accidental.

In my experience, the cure for harassment is a mix of a clear contract, impartial and neutral enforcement, education, and freedom. The most plausible punishment is exclusion (banishment), giving a perpetrator time and chance to self-correct, apologize, and prove their goodwill. Education means teaching people about the underlying models of harassment, to recognize distress in themselves and others, and to respond to it appropriately. Freedom means the ability to leave and form new groups, if authority is corrupt.

In the video, I guess the harasser is one of the louder students attacking the professor. He or she is using the situation to build up power over their peer group, and over the faculty. They're using the professor as an easy common enemy, to create emotions. They already have a slice of the crowd as followers. Tomorrow, a bunch more. Their goal is to use the entire campus as a power base. It's practice. They'll graduate to business or politics.

Ten Steps to Better Public Speaking

Speaking to an audience can be difficult for many of us. For the audience, it can be painfully boring. Yet when done well, a talk becomes a magical moment. I've had a few of these, among a lot of disasters. I'm going to explain my strategy for becoming a better public speaker. Some people are more gifted than others. No matter: the key is not talent, it is patient effort in the right direction.

Why It Matters

Once in 2007 I organized a conference on software patents, called EUPACO-2. We had a brilliant set of international speakers. The two keynotes were shockingly good, and for me, set the bar for every keynote speech I've ever given.

Mark Shuttleworth, dot-com billionaire and the first South African in space, told the story of the Internet revolution, and said a phrase I'll never forget: "*Old money hates new money, and tries to destroy it every chance it gets.*" His keynote inspired me to write on the digital revolution, and this fight between old and new money. That became Culture & Empire.

Our closing keynote, Bill Kovacic, was an ex-commissioner of the US Federal Trade Commissioner. He didn't use a microphone, and didn't need one. His voice boomed out over the audience. Two things he said, which are not in his slides, stood out. "*When Bush took office in 2000, every single anti-trust action in the USA stopped, dead.*" he said. And a little later, he explained why patent pools are so popular in large companies. "*Patents trump anti-trust law,*" he explained. "*When a group of CEOs get together to fix prices, that's a criminal offense and they will go to jail. But if they're cross-licensing patents, it's entirely legal!*"

Such power in so few words.

For me, this is the reason to attend a conference. It is to be kicked out of my inertia. It is to hear people with years or decades of experience lift the veil on mystery. It is to leave with words burnt into my mind, changing me, and pushing me to do things I'd never dare, otherwise.

It is a tragedy to see brilliance turned into a robotic slide reader. Kovacic had slides, for sure, yet what I remember is his hammering voice, his words, and his deep passionate anger at a corrupt system.

In technical conferences, the best track is often the corridor. I love the corridor, and the long fertile chats. Yet the corridor mostly doesn't discuss the talks. "Did you see so-and-so's talk? Did you like it?" is about as far as it goes. And then we're back to more interesting subjects.

It's better than nothing. A good bubble conference like BuildStuff, bringing people from far away and locking them up with booze and food, for days, can be mind blowing. It is like going on a cruise with a hundred close friends. A few weak talks, or many weak talks, is still worthwhile, if it brings people together.

Yet we can do better. I know we can do better and I think most of us who speak at technical conferences want to do better. Which brings me to this article and ten steps to better public speaking. I'm not going to pull my punches. It took me decades of public speaking to realize some of these techniques were even possible, let alone get good at them.

The Tips

1. Stop Using Slides

Slides are the training wheels of public speaking. They stop bad accidents that might scar you for life. Yet they are often the biggest barrier to improving your public speaking. Speaking without slides forces you to start learning the real skill of talking to, and not at, your audience.

Once in OSLO (NDC) I gave a talk on open source community building, and four people came. One was the sound engineer, one was an organizer checking on me, and two were there by accident. I gave the talk. The slides were, I think, excellent. My best slides ever, perhaps. And my last.

I've been giving public talks for a long time. Mostly they were the usual affair: twenty slides, five points each, weak delivery from behind a lectern. Good content, mediocre delivery, and a somewhat bored audience. And then sometimes, I'd give a brilliant, life-changing talk.

In June 2005 I came to talk at a small press conference-slash-debate organized by the European Patent Office and the FFII, fighting on opposite sides of the software patent war. This was the Christmas Football Match, a small moment of peace between the hostile camps. The EPO brought their usual suspects: bogus Microsoft-funded startups, industry lobbyists, and regulators. The FFII brought small businessmen, and me.

I asked to speak last. The two panels presented in turns. Each time the FFII's speakers were trying to show their slides, one of the organizers, from the EPO, kept adjusting the projector, playing with the audio and so on. It was not subtle.

When it was my turn I switched off the projector and said, "I'm not going to use slides."

My company had been fighting a software patent troll in the mobile app sector. I'd been talking with other mobile app firms to fight the troll. They all capitulated and paid. We refused, and the troll started attacking our clients. We had to shut down our platform. So I was motivated, and had good stories to tell.

I told off the fake startups for lying to the audience. I told off the EPO guy for trying to sabotage the other speakers. I told the audience, "patents don't create jobs. Hard work by quiet people creates jobs. Patents are a tax, a tool for lawyers to extort and blackmail.

Anyone who claims they need patents to write innovative software is a crook or a liar. We will fight this directive, and we will destroy it."

I spoke for fifteen minutes, unrelenting and angry. "You think you can come into *our* world of software, and change the laws so you can claim and tax our hard work?" We did crush the software patent directive (thanks to the hard work of others), and the FFII later asked me to become its president.

It was a lucky accident that day. Nothing to lose and enough anger to overcome my fear of talking without slides. For a decade that experience has been my benchmark and my goal. A lot has to come together to give a successful talk. No slides seems to be an essential starting point.

Let me break it down further:

- When you use slides, the audience watches the screen, not you. This reduces the strength of your communication. For instance your non-verbal body language is almost totally silent.
- Slides turn the audience from "participants" to "consumers." Half their brain switches off. They can always catch up later. They open their laptops. They think about their work.
- Slides interrupt your focus, as speaker. You switch from screen to audience and back. This switching is visible and makes the audience feel less important to you.
- A presentation, no matter how good, is fixed. There is no way to improvise, no way to adapt your talk to the audience in real-time. And yet that is how the best talks happen.
- Slides are an artificial device. We don't use them in real life, ever. No boy-met-girl story ever contained a 12 page presentation with bullet points. A successful sales pitch happens around food and drinks, not a meeting table.

I can go on. Enough to say that polishing your training wheels does not make you a better cyclist.

Let me quickly answer the common pro-slide arguments:

- "I need them to remember my talk" - it's a question of practice, like cooking without recipes.
- "They help the audience follow my talk" - if your talk doesn't make sense without slides, maybe you are fixing the wrong problem.
- "They make it easier to remember my talk" - visual experiences are entertainment. Learning needs words and voice, work and argument.

- "Organizers and audience demand them" - there are lots of conferences. Being a better speaker gives you more, not less choice.
- "It gives people a chance to catch my talk, later" - this is what the videos are for. If your talk can truly be captured on slides, why even be present in person?
- "I need a way to show data, source code, diagrams" - put them on line, and talk about issues that don't need such illustrations, rather than technical ones. Your talk will be far more interesting and useful.

We're so used to the crutch of slides that we've built our rituals around it. We treat our slot as a chance to dump data on the audience. And the audience expects it, groomed by years of mind-numbing meetings. Stockholm syndrome, perhaps. The audience pretends to understand and remember, and the speakers pretend to believe them. "Great slides! Where can I download them?" We all feel it's bogus yet can't quite verbalize it.

1. Remove the Barriers

When you talk to an audience, you must build trust rapidly. If you fail at this, everything you say will be rejected, no matter how plausible or well explained. Some audiences are more hostile than others. The majority will always be at least skeptical. The fastest way to build trust is to remove all props and barriers and show vulnerability.

Now we come to the deeper psychological reason for not using slides. It is about projection of power, the fixing of trust, and the opening of your audience's mind to your message.

In my work I've often had to resolve conflicts. There are some simple ways to build trust in a difficult meeting. You start by asking for token gestures, "could you pass me the water, please?" Then you remove barriers. Instead of sitting across the table from a person, you sit right beside them. Not so close to make them uncomfortable, yet within stabbing distance, so to speak.

Someone once described my speaking style as "vulnerable." This is indeed what I show, standing in front of the audience without any props to support me. And yet a show of vulnerability is not weakness. It is a claim of power. The other party either rebels, or accepts.

As Mark Rendle said once, this is like patting a large dog on the head. The dog can bite your naked hand. Yet if it does not, then it accepts your dominance. It looks like a friendly gesture, to pat a dog, yet it is all about power.

Slides are a prop. The moment you have something on the screen, you're not patting the dog's head, you're throwing it a stick. It is not the same thing at all. Watch an expert give a weak talk and you see this effect. The more they put on their slides, the less people believe them.

What other props do we tend to use? Obviously, the lectern. Don't stand behind anything. Walk down to the front of the stage and come close to your audience. If you can, get off the stage and get even closer.

How do you dress? Do your clothes make a statement? If so, they're a prop and working against you. I've learned to dress in plain, neutral colors. The less my clothes talk, the more the audience will accept my words.

How do you speak? Are you using jargon, making clever in-jokes, and referring to important companies and people? All of these are props, and they make you look weak. Learn to speak plainly, and if you tell jokes, make them about yourself.

Your own ego must disappear. The more you act or claim to be important, the less people trust you. The same goes for your technology or product. If you give a sales pitch, you are asking for rejection. Always talk from the audience's view. What problems do they encounter?

And here again, why slides are so toxic. How do you know the audience's experience unless you ask them? I like to discover this in the corridor, before my talk. Even a few conversations can give me the key to a talk. It is essentially unscriptable.

Don't use notes. I once had a last-minute panic, at a large event, and came on stage holding some small pieces of paper. The room was huge and filled, and instead of connecting to the audience, I connected with my hand. It was not a good talk.

1. Challenge Your Audience

Everyone is an expert in their own experience. Your best value as speaker is to ask the right questions, not provide the answers. As a layperson you can challenge an expert crowd. The best material you have is the data of your own experience. The best moments are entirely improvised, not rehearsed.

We come to the deeper question of how we *think individually, and in a group*. There is zero benefit in an exposition of existing finished material. "Here, I will now read the Wikipedia page on apple juice." Or worse, a corrupted version of that. "A fifty minute reading of my version of the Wikipedia page on apple juice." I call this Robot Reader Syndrome.

You must know the problems you're talking about in depth. This means collecting data from your daily experience, and using it to develop new crazy theories and models. It is just the scientific process: take real problems, develop wild answers, and get others to catch your errors. To present a finished product, with no space for negotiation or falsification, sterilizes the collective thinking process.

Let me give you an example. I've often seen conflict in open source communities. That's my data, and no-one can argue with it. Now I'll develop wild theories about where this conflict comes from, and how to solve it. That's my work. Then I present these wild theories to audiences who I hope have experienced the same problems. I wait, breathlessly, for them to tell me where I'm wrong. *Please tell me where my models are broken, so I can improve them.*

There are many ways to work on your material. Observe your own organizations and projects. Write, write, write. Talk with others, at meetups and in the corridors at conferences. Always, step away from selling an answer, and focus on identifying the real problems and challenges. While a Robot Reader is bad, a Robot Salesperson is worse.

If you are a good communicator, you can do this without even knowing the subject. It's how good consultants work. They enter a troubled organization. They ask staff, "what are the biggest problems?" They then ask others, "what are the best answers?" They write this down in a report, and charge a fat fee.

When you know your problems, you can learn to improvise around them, with time and practice. Don't give the talk people expect. Give the talk they need. Surprise your audience. I've done this a few times: propose one topic, then switch to a different one at the last minute. People mostly like that.

You may be tempted, as a speaker, to prepare a talk and then give the same talk several times. I've seen the same brilliant, interesting speakers give the same talk over months, even years. Part of this is the reliance on slides, again, which creates that sunk costs fallacy. "I've spent so long perfecting these slides that I might as well use them again." Part is the inability to improvise. Part is that culture of *ex-cathedra* monologues, the speeches and lectures.

Don't be that person who thinks or claims they're smarter than the room. No-one ever is. The only way to truly shine, in a roomful of people, is to be part of that room.

1. Learn the Technical Skills

Public speaking demands a set of technical skills. Some of us are naturally talented speakers. The rest of us can learn these skills with guidance and practice. This is a solved problem. One good solution: find your local Toastmasters club, and work through the program.

We are mostly better at walking than running. Yet almost anyone can complete a marathon, with practice and determination. We are mostly much better at talking to individuals than a group, yet anyone can learn to talk to a large audience.

There are a number of aspects to learn:

- How to speak clearly and slowly, without, uhm, filler noises. The most effective and brutal way is to record yourself speaking a text. Play it back, and note the problems. Too fast, too quiet, dropping sounds, hesitating, and so on. If you can't hear your own errors, get someone to help. As a general rule, speak more slowly, be more precise about the sounds, and lower your timbre.
- How to lose or shift your accent. Some accents are barriers, some are attractively vulnerable. It depends on the culture. In general, the more neutral you can sound with respect to your audience, the better. This can mean adopting the accent of your audience, or a neutral international accent.
- How to structure a talk. There are classic structures you can learn and use, e.g. introduction, three points, and conclusion. Or, take a classic story plot like overcoming a monster, a common theme in the software industry. My preference is for less structure and more dialog, which we'll come to later.
- How to time yourself. This is an essential basic skill. You must at the least offer the audience time for questions. It is a rude and inconsiderate speaker who takes all their time reading their slides, and then starts to break the schedule by continuing to talk. The conference should cut speakers off if needed, yet for me, self-timing is as important as not mumbling. Start by learning to talk for exactly five minutes. Then work up over years until you can structure a 50-minute talk on the fly.
- How to use your body language. This means stepping away from the lectern, and standing in front of your audience. It can be terrifying. Nonetheless, so is driving a car the first time. Good body language for talking is relaxed, open, and vulnerable. For example, arms open and using your hands, yet never raising your hands above your shoulders, nor making aggressive gestures.
- How to connect visually with your audience. Again, removing barriers is a first step. You cannot connect when you are reading your laptop screen. Then, speaking *to* specific people in the crowd. Speak to people close to you but not at the front. At the end of every phrase, switch to someone else, so you scan the room. If the room permits it, come down to the audience and literally talk to them. All the time, stay relaxed and smiling.
- Learn from Failure

You will always learn more from your failures than your successes. This means always pushing yourself out of your safe zone, and then using your stumbles to become a better speaker.

It can be intensely painful, humiliating, and discouraging to give a bad talk. When you aim for perfection and you set your standards high, you will fail (by your own measure) more often. The good part is that your worst talks will still be better than average.

For me, for example, a talk is a failure if I don't get more questions than I can answer in the time. My goal as speaker is to motivate people to do further research. So a cold room (or an empty one) is data that I did something wrong.

To avoid traumatic disaster, start with short talks in forgiving environments. Build up your confidence and skills over time. After every talk, do this:

- Get one or two points for improvement from someone in the audience whom you trust. "It was great," is not a help. "You spoke too rapidly," is helpful.
- Get someone to record you, and watch yourself on video. It can be unpleasant to watch yourself speak yet you need to do this to see yourself improve. It can often take months for conferences to publish videos, so don't depend on that.

Experiment with different formats so you build up your range. If you don't find events that offer you what you want, organize your own. For example, one of my favorite formats is a multi-day workshop, which I organize myself.

Watch other talks and identify what doesn't work for you. Be honest about your irritation and then ask whether you cause the same feeling when you talk.

When you do have a bad talk, use this as material. It makes a great opening, and I've done this. "Anyone here from Oslo? Yes? Well, I hate Oslo. Nothing personal, but last time I spoke there exactly four people came to my talk, and two of those had come to the wrong room."

Of course, it's never Oslo's fault. As speaker, you did something wrong, and you can fix that, and make it better next time. (My main lesson from Oslo was that the talk title makes all the difference.)

1. Maintain Protocol

Public speaking is an old art with a well-defined protocol. If a conference doesn't maintain protocol, organize it yourself.

I'm shocked by how few conferences know and maintain protocol. It's not complex, yet it does make a real difference. Here is how it tends to work:

- Speaker fiddles with laptop, while audience comes into room.
- After some extended silence, speaker asks, "everyone ready?" and starts talking.
- When speaker is done, audience applauds and walks out of room.

Here is how it should happen:

- The Master of Ceremonies (MC) stands on stage, watching the audience as they take their places.
- MC encourages audience to move closer, asks, "everyone ready?" and waits for an answer.
- MC welcomes the audience and thanks them for coming. He then introduces the speaker by name, and cites some of their accomplishments.
- MC asks the audience for a warm hand of applause, and welcomes the speaker up on stage.
- MC passes the mike, shakes hands with the speaker, and walks off.
- When the talk is over, MC asks the room to applaud, while speaker takes a bow (or looks bashful), and then MC takes back the mike.

This protocol isn't just decoration. It creates trust between speaker and audience, and it provides a fail safe in case the microphone isn't working (it's the MC who suffers, not the speaker). It is less stressful for the speaker. A good MC can save a flailing speaker from disaster, by intervening if the room is too cold to ask questions. And an MC can be a natural timekeeper, intervening when a speaker talks too long or bores the audience with endless slides.

If you organize a conference, please have an MC on every stage. If you are a speaker without such support, get another speaker to be your MC.

1. Build a Dialogue

Personally, both in the audience and on stage, I want a dialog. I want to be able to interrupt and challenge the speaker, and as speaker, I want the audience to do this. It is not about argument: it is about thinking together.

As speaker, you face Newton's First Law of Thermodynamics: people will sit quiet and not talk unless they feel provoked. So creating a dialog requires some planning and effort.

Start with the talk title. My best ever title was click-bait, "One Weird Trick for Making Perfect Software". I asked the organizer, is this acceptable, and she laughed, and told me it was great. And it was. A good title attracts and annoys, fifty-fifty. It fills the room with people who already want to argue with you.

Next, take control of the room. There are many ways to do this. It depends on the room, the audience, the culture, the time of your talk, and how much beer you had the night before. Here are some of my techniques:

- Watch the audience and make them wait, just a little too long, until there is total silence. This works in a larger room.
- Ask everyone to stand up and wait until literally the entire room is standing. Then ask them to sit again. I like to do this when I see people with open laptops.
- Ask everyone to stand up, and shake the hand of the person behind them. I've tried this twice in large rooms. One time the whole audience broke out in laughter. The second time, not a smile.
- Tell a self-deprecating joke about yourself. When you can get a room to laugh, you have taken control, as every comic knows.
- Start the talk by asking a simple yet difficult question, then give a reward to the person who answers correctly. I bring along copies of my books for this. It focuses attention and wakes everyone up.
- Thank the organizers again, and ask the audience to applaud. It's cheesy, yet it works.

When you've taken control of the room, you can start to talk seriously. You must anchor your talk in the experience of your audience. Again, I like to ask questions that show how common a problem is, or how rare a good answer is.

With large audiences (over 1,000) it can be helpful to bring someone on stage as your foil. If you follow my advice about having an MC, you have your foil. What the foil does is challenge you, in name of the audience. It is hard to chat with thousands of people, so the foil substitutes for the crowd.

I've also used a Twitter feed for questions, and that can work nicely. There is the danger of breaking the connection to your audience, as you read the tweets. The conference where I used that technique had arranged large screens at floor level, aiming up at the speaker, which was incredibly useful.

Every room is different, so you must spend time in the room and watch other talks in that same space if you can. In small rooms you don't need to pass microphones around. In larger rooms, you must. Or, you must repeat the question for the camera and rest of the room.

You can also change the layout of the room, in some cases. I've done this successfully in smaller events, where the seats started out in classic lines facing the stage, like a school class room. Ten minutes of moving stuff around, and we had a nice circle of sofas and chairs, almost like a campfire.

There are some terrible rooms: huge flat rectangles designed to hold a thousand people all sleeping at the same time. In such rooms, bully the organizers into switching off all video, then move left and right on the stage so you can speak to most of the crowd. You may also

kick off by asking people to move into the center, if the room is not full.

I like to poll for questions about half way through the time, and then drive the talk by answering questions. When this works, it is great. When it fails, it's dramatic. Recently there were no questions and so I simply stopped the talk early: the audience (like me) was exhausted from the party the evening before. Having an MC on stage is a good backup: I learned my lesson there, again.

Answering questions well takes skill at improvising. You don't actually need to answer the question. Rather, it can be an excuse for exploring another interesting topic. The best questions are a direct challenge to the ideas you are talking about.

1. Keep It Simple

This is perhaps the hardest skill of all: to explore one idea in depth rather than touch the surface of a hundred different ideas. Yet it is the difference between garbage and gold. Every piece of your talk should be carrying your story forwards, not telling other random stories.

There are logical (yet false) reasons why speakers try to cover way too much. They feel they have to fill their speaking slot (rather than give the audience time to think and argue). They still use slides, so work backwards from the time: "in one hour, I can convey ten big ideas." They suffer from the "ten slides good, twenty slides better" type of false reasoning (it works better as "ten slides bad, twenty slides worse.")

I think public speaking is like writing, carpentry, cooking, or music. You add value by making your product simpler and easier to digest, not by adding more. Any fool can make complexity. Simplicity is the real challenge.

The core of your talk should be an idea or argument or model that is rare, controversial, and valuable enough to justify the moment. It should be worth the time the audience takes to travel, and the cost to you and the organizers to be there. Above all, you should be answering real questions the audience is facing now or will face soon.

You can then explain and describe your model from many angles. These are not different stories. They are the same story, told in more and more detail. Each explanation gives the audience another perspective, and if you are doing your job well, they start to see the whole picture.

Do not expect people to understand a complex new idea in a single sitting. That is not how it works. All you can do is break the ice of skepticism and plant a tiny seed. If conditions are right, the seed will grow and many months or years later, come to fruit.

If you are challenging established culture, or authority, or habits, you will face resistance. The more investment in out-of-date models, the more resistance to change. Imagine this as wind. The simpler your idea and the more focused your presentation of it, the smaller your wind profile. And so, the lower the resistance.

1. Keep the Discussion Going

Your work starts long before you give a talk, and ends long after. Subtle and deep truths can take years to hit home. And your own thinking will evolve and deepen over time. So you help yourself, and your audience, by keeping the discussion going over months and years.

I use various techniques for this. Obviously, there's this blog and my books, which act as shorthands. Instead of repeating myself, I can refer to an existing chain of argument.

This article, for instance, came from a corridor discussion (well, more of a beer and rock music discussion) with Dylan Beattie. Perhaps in the future it would make a subject for a talk, in its own right. When you can, connect the past with the future like this. It just takes the habit of writing short pieces regularly.

However, articles are opinion. Facts are more compelling. So a key part of the long term discussion is code and formal documents such as RFCs. These are not truth, yet they are easily falsifiable. If no-one uses a particular piece of code, or a given contract, you can assume they are inaccurate or irrelevant.

I also like to collect my videos, as it gives people a view that stretches over years. It is also about self-promotion, something you must do as a speaker. People need to want to come to my talks. Empty rooms are no fun for anyone.

Self-promotion sounds negative, yet it's essential. Partly you need to build and understand your own success, to be a better speaker. This is to deal with imposter syndrome, which I'll explain in the next section. Partly, you need access to interesting conferences, and that means gaining a reputation and a following.

My advice is to build your reputation by your work, no more or less. The world cares about results, not ideas or vision, or even history. Don't tell us who you worked for, who your best friends are. Tell us what you made and give us a link so we can check it out.

If you are, like me, in the software business, *invest in open source*. It is the number one way to build your knowledge, and reputation. Your Github profile is your CV. This is mostly true today and it is almost entirely true tomorrow. Show me a solid history of work and I trust what you say. Show me a blank page, and I'm going to start questioning your motives.

Once you drink the github koolaid, you can use it for everything. Put your talks there, and accept pull requests. Put all your writing there, and accept pull requests! Use it for your blog (and accept pull requests).

1. Manage Your Emotions

Lastly, and hardest for many of us, is dealing with the fear and anxiety of speaking to a crowd of strangers. Many of us don't feel competent, or confident enough. We are afraid of saying the wrong things, of looking foolish, and of being rejected. Yet there are ways to overcome all these emotions.

The fear of performing is called "imposter syndrome." It affects many speakers, as it affects many people who put themselves in front of an audience. The feeling is one of, "I don't deserve to be here," and it can be crippling. It is worse for anyone who suspects they were invited for reasons other than their skills and experience. It is often worse for women than for men.

All our emotions can be tamed, it is a matter of technique and practice. I've written about this on my blog and in my book The Psychopath Code. This also applies to imposter syndrome, which is essentially the fear of exposure as a fraud, and rejection.

The fear is based on a chain of subconscious assumptions. One, that we're not really good, just lucky. Two, that we will stumble, and reveal our incompetence. Three, that people will react by rejecting us in horror.

Let's tackle competence first. This is, I feel, a symptom of society placing too much burden on the individual to be special, and different. Accept that we're all ants, little pieces of a complex system that works astoundingly well. Sure, we can flicker with individual brilliance now and then. It means little. Our superpower comes from working together. Simply by existing, you add value.

Next, fear of failure and exposure. This is a symptom of culture asking us to be perfect. I've said, embrace your mistakes and learn from them. Fail with happiness and grace, recover, and continue. When there is nothing to hide, there is nothing to expose.

Last, fear of rejection. There's a small mantra you can repeat to yourself. It's our vulnerabilities that make us attractive to others, not our strengths. We reject people who are difficult to work with because they are anti-social, arrogant, and deceptive. Such people never feel imposter syndrome. The simple fact you're afraid to fail makes you precious to others.

Here are other techniques I recommend to fix imposter syndrome:

- Remove the barriers between you and the audience. It feels more scary at first and then you realize that the less you try to hide, the less fear you have of exposure.
- Treat failure as science. It is better to try ten things, and see eight fail, than to try just one and hope it works. For me, this means starting lots of projects, writing lots of articles, and speaking at lots of talks. Many are failures. The ones that survive are my

successes.

- Before the audience comes into the room, go onto the stage, and walk around. Tell yourself, "this is my stage, this is my place, and the audience are my guests and friends."
- When you talk, never lie, bluff, or make claims you can't back up. When you don't know something, just say, "I don't know." There are people who are professional liars: you're not one of them.
- Stay busy. If you have a talk coming up, write an article about the topic. Keep researching it, and try out different story lines on people.
- Trust yourself. You are the child of a billion generations of survivors. Good luck has nothing to do with it. You have winning genes backed up by hard work.

Conclusions

If you're passionate about your work, then sooner or later you will want to share that passion with others. There are many reasons to want to become a good public speaker. Perhaps the main one is simply to get out into the world and meet interesting people. For me, it is a way to learn from others, and shake up my ideas in a way that's impossible working alone.

I've explained my best techniques for capturing an audience's attention, and getting them to consider your ideas seriously. It is not easy. The typical conference attendee remembers only 9% of what they hear, by the end of the week, and a month later that figure is down to 0.1%. OK, I'm making this up, yet you get my point.

Stop using slides, get closer to your audience, and challenge them. Engage them in dialog, and make them think. Keep your voice and body focused on the room. Use no props, gimmicks, or notes. Leave your fear off-stage, and if a talk bombs, accept that with grace.

It is a slow, long process, so be patient and always kind to the organizers and staff.

Chapter 4. A Protocol For Dying

I could probably write more, yet there are times for everything and after this, my attention will be focused on the most comfortable position for my bed, the schedule for pain killers, and the people around me.

Yesterday I had twelve visitors, including my lovely young children. You'd think it's exhausting, yet the non-stop flow of friends and family was like being in a luxurious hot bath with an infinite supply of fresh water.

I was a disconnected and lonely young man. Somewhat autistic, perhaps. I thought only of work, swimming, my pet cats, work. The notion that people could *enjoy* my company was alien to me. At least my work, I felt, had value. We wrote code generators in Cobol. I wrote a code editor that staff loved because it worked elegantly and ran on everything. I taught myself C and 8086 assembler and wrote shareware tools. The 1990's slowly happened.

Over time I learned that if you chat with a stranger, in the course of any kind of interaction (like buying a hot dog, or groceries) they'll chat back with a beam of pleasure. Slowly, like a creeping addiction to coffee, this became my drug of choice.

In time it became the basis, and then the goal of my work: to go to strange places and meet new people. I love the conferences because you don't need an excuse. Everyone there wants, and expects, to talk. I rarely talk about technical issues. Read the code, if you want that.

And so I'm proud of my real work, which has been for decades, to talk with people, listen and exchange knowledge, and then synthesize this and share it on with others. Thousands of conversations across Europe, America, Africa, Asia. I'll take whatever credit people want to give me for being creative, brilliant, etc. Yet the models and theories I've shaped and documented are consistently drawn from real-life experience with other people.

Thank you, my friends, for that. When I say "I love you" it's not some gesture. You literally kept me fed, professionally and intellectually.

So I wanted to document one last model, which is how to die, given some upfront knowledge and time. I'm not going to write an RFC this time. :)

How it Happened

Technically, I have metastasis of bile duct cancer, in both lungs. Since February I've had this dry cough, and been increasingly tired and unfocused on work. In March my Father died and we rushed around arranging that. My cough took a back seat. On April 8 I went to my oncologist to say that I was really not well. She organized a rush CAT scan and blood tests.

On 13 April, a horrific bronchoscopy and biopsies. On 15 April, a PET scan. On 16 April I was meant to drive to Eindhoven to keynote at NextBuild. Instead I went to the emergency room with explosive pains in my side, where they'd done the biopsies. I was checked in and put on antibiotics, which fixed the pain, and on 18 April my oncologist confirmed it was cancer. I'm still here, and my doctors are thinking what chemo to try on me. It is an exotic cancer in Europe with little solid data.

What we do know is that cholangiocarcinoma does not respond well to chemotherapy. Further, that my cancer is aggressive and fast moving. Third, I've already some clusters in other parts of my body. All this is clear and solid data.

So that day I told the world about it, and prepared to die.

Talking to a Dying Person

It can be horribly awkward to talk to a dying person (let's say "Bob"). Here are the main things the other person (let's say "Alice") should not say to Bob:

- "Hang in there! You must have hope, you must *fight*!" It's safe to assume that Bob is fighting as hard as possible. And if not, that's entirely Bob's choice.
- "This is so tragic, I'm so sad, please don't die!" Which my daughter said to me one time. I explained softly that you cannot argue with facts. Death is not an opinion. Being angry or sad at facts is a waste of time.
- "You can beat this! You never know!" Which is Alice expressing her hope. False hope is not a medicine. A good chemotherapy drug, or a relaxing painkiller, *that's* medicine.
- "There's this alternative cure people are talking about," Which gets the ban hammer from me, and happily I only got a few of those. Even if there was a miracle cure, the cost and stress (to others) of seeking it is such a *selfish* and disproportionate act. With, as we know, lottery-style chances of success. We live, we die.
- "Read this chapter in the Bible, it'll help you." Which is both rude and offensive, as well as being clumsy and arrogant. If Bob wants religious advice he'll speak to his priest. And if not, just do not go there. It's another ban hammer offense.

- Engage in slow questioning. This is passive-predatory, asking Bob to respond over and over to small, silly things like "did I wake you?" Bob is unlikely to be a mood for idle chitchat. He either wants people close to him, physically, or interesting stuff (see below).

Above all, do not call and then cry on the phone. If you feel weepy, cut the phone, wait ten minutes, then call back. Tears are fine, yet for Bob, the threat of self-pity looms darker than anything. I've learned to master my emotions yet most Bobs will be vulnerable.

Here are the things that Alice can talk about that will make Bob happy:

- Stories of old adventures they had together. Remember that time? Oh boy, yes I do... it was *awesome*!
- Clinical details. Bob, stuck in his bed, is probably obsessed by the rituals of care, the staff, the medicines, and above all, his disease. I'll come to Bob's duty to share, in a second.
- Helping Bob with technical details. Sorting out a life is complex and needs many hands and minds.
- "I bought your book," assuming Bob is an author like me. It may be flattery, or sincere, either way it'll make Bob smile.

Above all, express no emotions except happiness, and don't give Bob new things to deal with.

Bob's Duties

It's not all Alice's work. Bob too has obligations under this protocol. They are, at least:

- Be happy. This may sound trite yet it's essential. If you are going to be gloomy and depressed, Alice will be miserable every time she talks to you.
- Obviously, put your affairs in order. I've been expecting death for years now, so had been making myself disposable wherever I could. For family, that is not possible. For work, yes, and over the years I've removed myself as a critical actor from the ZeroMQ community.
- Remove all stress and cost that you can. For example Belgium permits euthanasia. I've already asked my doctors to prepare for that. (Not yet!, when it's time...) I've asked people to come say goodbye before I die, not after. No funeral. I'll give my remains to the university here, if they want them.

- Be realistic. Hope is not medicine, as I explained. If you are going to negotiate with your doctors, let it be pragmatic and in everyone's interests. I've told mine they can try whatever experimental chemotherapy they wish to. It's data for them, and the least I can do for a system that's given me five+ years of extra life.
- Assume the brutal worst. When my oncologist saw my scan she immediately called me and told me, in her opinion, it was cancer. In both lungs, all over the place. I put the phone down, and told the children. The next day I told their schools to expect the worst, then my lawyer, then my notary. Ten days later the biopsies confirmed it. That gave us ten more days of grieving and time to prepare.
- Be honest and transparent with others. It takes time to grieve and it is *far* easier to process Bob's death when you can talk about it with Bob. There is no shame in dying, it is not a failure.

Explaining to the Children

My kids are twelve, nine, five. Tragic, etc. etc. Growing up without a father. It is a fact. They will grow up with me in their DNA, on YouTube as endless conference talks, and in writing.

I've explained it to them slowly, and many times over the years, like this. One day, I will be gone. It may be long away, it may be soon. We all die, yes, even you little Gregor. It is part of life.

Imagine you have a box of Lego, and you build a house, and you keep it. And you keep making new houses, and never breaking the old ones. What happens? "The box gets empty, Daddy." Good, yes. And can you make new houses then? "No, not really." So we're like a Lego houses, and when we die our pieces get broken up and put back in the box. We die, and new babies can be born. It is the wheel of life.

But mostly I think seeing their parent happy and relaxed (not due to pain killers), and saying goodbye over weeks feels right. I am so grateful not to have died suddenly. I'm so grateful I won't lose my mind.

And I've taught my children, to swim and bike and skate and shoot. To cook, to travel and to camp. To use technology without fear. At three, Gregor was on Minecraft, keyboard in left hand, mouse in right. At seven, Noemie learned to shoot a pistol. They speak several languages. They are confident and quick learners, like their dad.

And everyone needs to learn what it means to die. It is a core part of being a full human, the embrace of one's mortality. We fight to live, of course. And when it's over, we embrace the end. I'm happy that I can teach this lesson to my children, it is one that I never had.

Euthanasia

I am, finally, so glad I never quit Belgium. This country allows for death on demand, for patients who are terminal or have a bad enough quality of life. It takes three doctors and a psychiatrist, in the second case, and four weeks' waiting period. In the first case, it takes one doctor's opinion.

My dad chose this, and died on Easter Tuesday. Several of us his family were with him. It is a simple and peaceful process. One injection sent him to sleep, into a coma. The second stopped his heart. It was a good way to die, and though I didn't know I was sick then, one I already wanted.

I'm shocked that in 2016 few countries allow this, and enforce the barbaric torture of decay and failure. It's especially relevant for cancer, which is a primary cause of death. Find a moment in your own jurisdiction, if it bans euthanasia, to lobby for the right to die in dignity.

My Feelings on All This

I've never been a fearful person. My last brush with death left me so casual about the whole concept of professional and social risk that I became the predatory character Allen Ding so nicely describes. That calmed down after our Game of Thrones project ended. It was never really me, just the person I became to make things work, in that place and time.

Having had years to prepare for this, and having seen a great many delicate plans come together over those years, leaves me deeply satisfied. Since 2011 I've become an expert pistol shot, taught myself to play piano (and composed many small pieces), seen my children grow into happy, bubbling characters, written three books, coached the ZeroMQ community into serene self-reliability. What more can a Bob ask for?

The staff here are lovely. I've no complaints, only gratitude to all my friends for the years of pleasure you've given me, my drug, which kept me alive and driven.

Thank you! :)

Think of the Children

Please use this article to add your stories. If you have them elsewhere, or you emailed me, copy/paste as a comment. Feel free to write in Dutch or French if that's your language. I'd really like a single place where my kids can come and read what other people say about their dad.

Chapter 5. Planned Death

On Monday I got home from the hospital. It was nice, as hospitals go. My sisters and mother came to get me, and after hours of paper work and last minute urine tests, the doctor gave me my papers and told me I was free to leave. Without oxygen, I sat quietly during the taxi drive home and then got home.

As I sat down in my living room, surrounded by the material of my life, my children, my family, my cats, I cried. The overwhelming sense of joy was so intense that I almost started tripping. Then I found the oxygen, and things came back into focus.

The chemo, this first round, was miserable. I wanted to write a happy note about the lack of side effects, to cheer people up. Instead I spent most of Friday and Saturday vomiting, in one case projecting green-yellow vomit across several meters of my room. I think I'd eaten a couple of olives for supper. Hospital rooms and staff take this kind of thing in their stride. A few sweeps and it was all gone.

I'm not a good vomiter, and the doctors gave me increasing doses of different drugs to fight the nausea. Only when my oncologist said, "if you vomit again, we'll stick a drip back into you and start brain scans," did my body realize this had to stop. It's like an anti-placebo. The threat of not getting free, and worse, switched off all my nausea like flicking a switch.

Curious to see how it goes with the next round of chemo next week, now that I know the effect is at least partly psychological.

Vomit or not, I'm dying. I'm on palliative care, and the doctors did not switch off their slightly grim "we'll do what we can" manner. So different from their "oh, you'll be fine!" faces.

To be clear, I'm not resigned, hopeless, or fatalistic. I'm *absolutely* determined to beat this cancer, by sheer force of will, and blind hope in the miraculous. This is how I'm designed, like a unbreakable self-righting toy. Put me into any situation, no matter how impossible, and I will always find a way to make things work. Yet I know that I'm bullshitting myself in this case. "Always" only ever means, "so far, so good."

Status: awesome :)

I can feel the cancer growing in me, my breathing slowly getting shallower and less easier. It is like the atmosphere has lost its essence. I breath oxygen through a tube, one liter a minute, and live with this cable like in the old times, when computers could move as far as their wires. A portable saturation meter (thank you, Amazon, you've made my life so much easier over the years) tells me I'm at 96%.

Sometimes I take off the nose piece, and "go WiFi" for a few minutes, maybe half an hour. This lets me put my sons to bed, tell them stories. Gregor, the youngest, is so comfortable in my arms he falls asleep after two to three minutes, always.

Freeman listens to my stories with attention, as he has always done. Over the last years we told the stories of Bobolan the Magician, who built the largest magical university ever. It all started at the End of the World, when Bobolan had to find five precious stones to restart the Sun. That story took us a month to tell. Jack Vance, Terry Pratchett, forgive me for taking a few riffs from your songs.

My children seem to be doing well. They are calm and cheerful, focused on their daily rituals. School, computers, feeding the cats, helping in the house. It took a day or so for us to adjust after my absence and return.

There is no hiding that I'm sick: my medical bed with its motors and cables is in our large living room, the center of our home. When I cough, my kids hear it, they can follow my slow downwards curve.

Not that I'm getting weaker. Hospital left me weak and thin, 14 kilos lost, slow shuffling walk. Now I am quickly getting back in shape, up and down the stairs, WiFi. Being at home means I feed myself. Food has taste again!

And home, I have space for my friends and family as they visit. I hope the funerary procession of that Sunday ten days past is finished, when fifteen people visited me in my small room. Some I'd not seen for years, decades. Why, I wonder, would you come to see me sick, when we don't talk in real life? I get it though, the social rituals of saying goodbye. I notice that flexibility, being able to greet the not-yet-dead in much the same way as observing silence in presence of a body.

A friend came up from Paris with a bottle of wine, warm socks, chocolate drops she made in her restaurant. My daughter ate the chocolates. I put the socks on. We opened the wine. Wine and beer, I asked my doctor, before checking out. How much can I drink? He frowned... if you're not taking medication, then a glass or two can't do any harm.

She told her colleagues, "I'm taking a few days off work to say goodbye to a friend who's dying." A strange reason, perhaps, and yet spending a couple of days together, slowly talking about life, knowing this is the last time we'll see each other... it feels as natural as cooking a meal with friends.

I think "euthanasia" as a term has some problems. It's too easily hijacked by the "death panel" lunatic fringe who believe that pain and suffering is our destiny. Switch off your Internet and heating, you psychopaths, if that's what you really think. No, what they really mean is "I claim the authority over others."

Let's use the term "planned death," it is accurate. Planning our deaths. It may be a luxury for some, and for others, besides the point, yet for those like me who see the road ahead, I claim this to be an essential Human Right.

How else can this all work? My children don't want or need to see me rot and fall apart. They don't want me to go away again, disappear into that machine called "medicine." Talk about a lifelong trauma. I can offer my children a model of control, careful organization, order out of chaos. It's what I've always taught them. Chaos is the default: do not wait for others to fix it. That's your responsibility: organize your world, take control.

Not all doctors are willing to kill their patients. It is the first discussion I had with my new family doctor. "Are you on board with the whole killing me when it's the right time thing?" Sure, he said, your oncologist already asked me that before choosing me. Good man. We signed a contract and shook hands.

My oncologist. I have to say, a wonderful woman. I joked with my sisters, when the tall male head oncologist visited me in my room, that I was grateful to finally see a *real* doctor. My sister, a doctor, pretended shock and outrage. In truth, my oncologist took me under her wings and pulled every string necessary to get me home in the best of conditions.

A planned death is not a moment in time, like a car accident or a fatal stroke. It is a process. A social process that involves hundreds of people, each doing their part, grieving their loss, accepting their own mortality.

I'm proud and immensely grateful to be able to experience this, and share it with you.

Chapter 6. So Far, So Good

It's been two weeks since my last update. People ask me, every day, "how are you doing?" so I figured it's time to sketch this out in more detail.

There's good news and there's bad news. The bad news is that I'm weak, and not getting stronger. A few times every day I find my lungs choking up, and I cough to clear them, until I'm vomiting. It's not pretty. We get used to it. My son Gregor puts down his Splatoon game and pats me on the back, as I retch into my bowl.

The chemo is, I'm sorry to report, awful. I had wanted to tell you it was mild, a minor blip, yet I'm spending four of five days every two weeks, messed up.

At least I'm not losing my hair. Instead, I get cramps in my esophagus and jaws whenever I try to eat, or move. This lasts from Thursday to Saturday. In between the ache and pain of my chest feeling like I've tried to swallow too much dried bread, I sleep, or vomit.

I tried to walk a little back from the clinic, and ended wheezing like an old man, desperately looking for places to sit while I caught some strength back.

The good news is that once this is past, by Monday or so (the chemo is on Wednesday and lasts 48 hours), I'm pretty good.

Today I took my boys to school, by bike. I cycle slowly in first gear. In Gregor's school the staff greet me with that mix of emotions I'm used to. "Please stop wishing me courage," I want to tell them. Just smile and relax. The emotions can be overwhelming.

My blog post, "A Protocol for Dying" went viral. That was a surprise to me. The last time one of my posts went viral was about ten years ago when I wrote on Slashdot, "War Declared on Caps Lock Key".

That time, like this time, my raw, unedited expression struck a chord and was picked up. First by the on-line community. The Slashdot article got massive commentary. My Protocol got massive attention on Hacker News.

Then by a few braver journalists, and then by the mass media. This time, though, it went further. I was on Belgian television on Sunday, and on another program this evening.

Of course I love the attention. For years I've worked to promote myself as a product. Hintjens, the guru. Ironic that dying turned out to be the greatest marketing stunt of my life.

Two small yet significant things came into my life these last days. At heart I'm still a young boy, fascinated by technology and gadgets. In the hospital I'd dropped my phone, and the microphone stopped working. So for weeks I've been plugging in an external headset, shouting "hang on, hang on" to the caller, as I fumble through tangled cables.

Then a package arrived with a Xiaomi Redmi Note 3. Magically it made its way through customs without delay, coming from Hong Kong.

It's just a phone. A nice one, well executed, almost impossible to fault, and cheap. Yet what it represents is the victory of Chinese open source hardware over all competing models. If you've not yet understood the Shanzai model that drives Chinese innovation in industrial production, look at it. It is the future.

I'll describe it briefly. Every firm in this culture publishes their Bills of Materials, and design specs. Any other firm may take these, reuse them, improve them. They must also share back.

The Shanzai model originated in the 1990's when Chinese electronics were nasty imitations of western and Japanese products. It has grown into the dominant model for pretty much every industrial product (not just electronics) produced by Chinese firms.

This is why you can buy the same product from a slew of firms on Amazon or Ebay. This is why the price drops smoothly, as predicted by Cost Gravity. No patents and trade secrets to slow down the spread of knowledge. This is why Chinese products haven't just caught up to western designs. They are way, way ahead. My Xiaomi is built of 95% Chinese components. This is why Apple will die.

The second gadget I received is a funny thing called the Freewrite. I'm using it to write this article now. I can't yet tell whether the Freewrite is entirely insane, or genius. It all depends on how Astrohaus, the firm backing it, can deal with its users.

This is an American design. It focuses on writing, not editing. There is no editing except backspace. (One character, one word, or one paragraph.) It is heavy and clunky. It has a slow, small e-ink screen that always lags behind. It has no ports except one USB C. There is little documentation, no explanation of its internals.

And yet I love it. I wrote a review the first day I'd gotten it. 9/10 was my feeling then, and today that still stands.

When I wrote my rant against the Caps Lock key, it was because I'd just received an AlphaSmart Dana. This is literally the last product in the same niche as the Freewrite. The Dana had a lovely keyboard and ran a weird widescreen PalmOS. It was instant-on, daylight readable, and wonderful for stream-of-consciousness drafting.

And it had a bloody Caps Lock key that could not be disabled. That creates a dead zone right where the Ctrl key should be. Touch that dead zone and your writing turns into zombie capitals. Such an irritation it was that I wrote my rant on Slashdot, and for a few weeks, became famous.

The Dana never got updates. The company that made it could not deal with mass market success. PalmOS died. It was so tragic, as the Dana and its predecessors were so *perfect* for the job, except that slight bother of lacking any support at all.

For ten years I waited for someone to produce a replacement for the Dana. Please, give me an electronic typewriter with long battery life and sunlight readable screen. I'll pay for it. Use e-ink, a good keyboard, and keep it simple.

And then Astrohaus launched their Kickstarter and I sent my money and prayed for this thing to even half work. Overpriced, some people complain. I shake my head. You don't get it. We waited a decade for this. It is literally the only living product of its kind in the world, possibly the universe. Please, Astrohaus, take my money and do good things with it. Don't die like AlphaSmart.

So it finally arrived, just in time for my extended home stay. And bingo! A huge ominous Caps Lock key that can't be switched off. I guess it's a tradition. At least Astrohaus are promising firmware updates and have acknowledged my plea to turn this key into something more useful.

Speaking of blasts from the past, last night my kids and me watched The Magnificent Seven on Netflix. The film's slow and careful pace confused my kids at first. Where is the drama, they wondered. Then Kurosawa's brutal story slowly unfolded, and transfixed them. At some point Steve McQueen's character describes a man falling off a tall building. "So far, so good," people hear him say, as he falls.

This is how it feels. So far, so good. There is a crash coming. I can feel it, my lungs and chest squeeze in strange and uncomfortable ways. My body needs too much sleep. I'm not recovering strength. No pain, apart from the occasional spasms around my esophagus. I stopped using oxygen last week and it went fine. That is so great, not having to stay tethered to an oxygen cable all the time. So far, so good.

Speaking of which, we're getting ready for our party-slash-wake-slash-riot on June 5th. Fifty extra chairs are on their way. We have found a place to supply the grilled goat meat. (Seriously, have you *never* tried that? With fried plantains, murderous hot sauce, and cold beer.) We are freezing ice for the beer and preparing the stage for the DJs.

Venez nombreux. It will be fun. And it looks like I will still be in good enough shape to take part.

Now you will have to excuse me, my kids just got back from school and that means family time. :-)

Chapter 7. Living, In Limbo

I've written a lot in my life. This is perhaps the most difficult piece I've ever done. In April, with a diagnosis of terminal cancer, I prepared to die. It's now July, and I'm still not dead. Instead, I'm in an in-between state, neither healthy nor obviously sick. Limbo is a strange land.

The Undead

"Is he getting better, or is he dying?" asked my nephew of me. How to explain? The hospital sent me home three months ago with boxes of pain killers, oxygen, a medical bed, and home care. Palliative care: aim for quality of life, not return to normal. And yet here I am, not on oxygen, not taking the pain killers, and seeing medical staff only when it's time for my biweekly chemotherapy.

I'm clearly not dying yet. And still, slowly losing weight and muscle. A simple walk leaves me tired and needing to sit. I wake up, make an early morning cup of chicory/coffee, drink it, then lie down again, hit by the simple effort of standing up.

We did a CAT scan a few weeks ago. Inconclusive. Things don't seem worse. Yet the numerous little blobs of cancer are still there in my lungs, patient. Another scan in a month, and we'll have a better idea.

At least my horizons have grown. When I wrote "A Protocol for Dying" I was convinced that only a month or two remained. We had our wake-slash-party at the start of June, and I was there, not in a coffin, but sitting up alive and well. And playing a little with the musicians, even.

Months, then. Maybe even a year, if we see the chemo is holding down the cancer. A year. It is immeasurable. I don't hope, just observe.

My daughter asked me, "can we go camping this year?" The kids and me, loading up the car and driving off to far places. It became a tradition. We're a gang, well-organized and self-sufficient, as long as we're within reach of a Lidl or Aldi.

Camping. Mostly we do the Atlantic coast of France, the islands d'Oleron and Re, Piccardie. Last year, the maritime Alps, in between Gap and Aix. Perched high on a hillside, we survived wind and rain when it came, and basked in the sun when it shone.

Putting up a tent seems an insane proposition, when I can barely climb stairs without losing my breath. So I called our mountain side campsite and the owners cheerfully suggested a bungalow with handicapped access. Ah, wonderful.

I asked my oncologist. Is it insane to skip a round of chemo and drive two days to southern France with the kids? "Great idea!" she said. "You must aim for quality of life!" No risks? I double checked. "You'll be fine," she said, "just enjoy yourselves."

It's on a mountain side, this campsite. Sometimes I think I've gone crazy.

The Chemotherapy

Half the reason it's so hard to judge my health is the chemo. I'd wanted to write, "meh, chemo is OK, nothing to be worried about." That's what I wanted to say. A lot of people who have cancer are afraid of chemo. It's got a reputation for messing you up.

The chemo messes me up. It doesn't make me bald, which is one consolation. Rather, there is a solid week of raw fatigue, vomiting, and distaste. Chemo day is Wednesday and it takes me until the next Tuesday to get over it. There are no drugs that help. Just sleep and time.

Yet if I'm writing this, it's because the chemo is doing something positive. That, or my immune system has suddenly kicked into hero mode. I should be dead. I'm not.

Lesson is: take your medicine. It may hurt, yet the alternative will hurt more.

I trust my oncologist with my life. Rather, I trust the medical machine, representing science. Live or die, my case forms part of that machine, data of succcess or failure. Can we treat bile duct cancer with Folfox, a combination of drugs developed for colon cancer? "Treat" is relative. If the chances of survival go from 5% to 10%, that's a big success.

The Party

When my Czech and Slovak friends drove up with a car filled with beer and meat, I knew this was going to be a unique weekend. On Saturday we had a meetup for our ZeroMQ community, which grew as the beer flowed, the barbecue sizzled, and more and more people turned up.

On Sunday around a hundred people came, all precious friends and family. The tables creaked with food. Three barbecues worked, all afternoon. We ate, talked, drank.

A group of musicians turned up and began to set up on the stage. Then a dancer put on a tape and started teaching moves to the crowd. Soon we had a dance class going. More musicians turned up. They started to play, and it was excellent. I didn't realize we had

several gifted musicians in the audience as well. The jam started. Guitar and lyre.

The amazing thing about this weekend was how smoothly everything went. Every problem had someone in charge of it. The beer stayed cold, the barbecues sizzled, the music played.

I've given many, many parties in my building, which is graced with a large space designed for this. There are a lot of things that can go wrong, from power failures to full on fights. I've had to chase out thieves and drunks, call the police a few times, apologize to the neighbors for the noise, broken bottles, and worse.

None of this happened. Instead, we finished at a decent hour, and the last people there cleaned the place up.

This was a classy party, with people I'm proud to consider my friends and to welcome into my home. Best party ever!

The Writing

You'd think that I'm in the ideal position to write. Lots of time, mostly at home, no long term plans. And yet it has been hard. It's taken me a month to start on this article. In limbo, it is so much easier to just switch off, become passive. It doesn't matter anyhow, does it. Just that constant prodding from my friends: "Pieter, how are you doing? What's your status?" And it's easier, eventually, to explain properly, than to answer in drips.

It is a challenge, this situation. Today is a good day, yet I wake up choking, coughing to clear my lungs. A voice tells me, Pieter, it's not getting better. And then another voice comforts, hey hey, the pain in your shoulder has gone. You're not in pain. You are eating. This is awesome!

There's this book, Scalable C, which I want to work on. It sits there, accusing me. "You promised!" it complains. I know, I know, I reply. Then back to Hacker News and Reddit. Is it the chemo, that's changed me? Or is this what it's like, in Limbo? I can't tell.

The World in Limbo

2016 is turning out to be a strange year for many people. Watching the Orlando shooting, the bombing in Istanbul, the attacks in Dhaka, the US elections, and the British dipping their feet into the sea of madness, I can't help but see patterns.

Seems to me, all these events follow the same underlying pattern. You can decode it from basic principles. We see mass pain and suffering, exploited or provoked by a few individuals. Sometimes there is a long term goal: political power, often. Sometimes it's

nothing more than "taste blood before I die."

The ability to hurt many for personal gain is exclusive to those people we call psychopaths. Most psychopaths are well aware of the costs of getting caught, and work hard to avoid that. In rare cases this mechanism stops working. The higher the stakes, the more we'll risk. Was the Orlando shooter feeling worthless and suicidal? If so, all restraint is lost. Does the concentration of power in the UK and US drive politicians to risk everything? If so, they become like mass killers, yet on a national scale.

All human behavior has, I deeply believe, motivation that can be decoded, tested, and shifted. Religion does not make men mad. It lies about the economics of crime and punishment. Like the lies we saw in the UK referendum. A mind makes insane decisions when given false data.

My prediction for the UK is, incidentally, that there will be a new prime minister, then a free vote in the Parliament, steered by the referendum, and its consequences. The UK, advised by truth, will step back from madness, will remain in the EU, and will not break up. The experience will make Europe stronger. Enough with the nationalism. Maybe, a new generation of British politicians will finally address the economic and social poverty that has hit England since Thatcher and Blair. Maybe.

Thank you

I'm truly grateful to so many people for their help and support over the last months. Thank you for coming to Brussels and for visiting me, for buying my books, the single malts, for sending me money, for writing to me, and being there.

Chapter 8. Fighting Cancer

There are no easy conversations when it comes to dying. Especially when it comes to a disease like cancer, which eats us up from the inside, a betrayal by our own cells. "Fight it," people still tell me. "Don't give up! We need you!" This notion that cancer is a fight... it's one I want to break down, and then rebuild, in this article. I've come to believe that death can be a positive social experience. Let me explain...

Let me start with this: one does not choose to fight, or give in to, a disease like cancer. Perhaps to any disease. In my body right now there is a holy war going on, and has been raging for years. My immune system has been doing its damned best to kill these rogue cells. And the rogue cells, unaware that they're destroying their own host, have been fighting back. It's no small fight. I've lost 15 kilograms in the last few months.

The odds are on the cancer, of course, which is why this family of diseases is a major killer. Our bodies have to keep winning, year after year. Any given cancer has to win only once, and it's Game Over. The only way to beat cancer, really, is to die from something else first.

So this is my first point. Everyone fights cancer, all our lives long. From birth, our immune systems are hunting down and killing rogue cells. I grew up in the African sun, pale skin burned dark. Do I have skin cancer? No, thank you very much, immune system! Much of my adult life I drank a bit too much, ate too much red meat, too few vegetables. Do I have bowel cancer? No, thank you again, you over-active beast of an immune system, you! Hugs.

And most of us can say the same thing, most of the time. We are all cancer survivors, until we're not.

Secondly I want to attack that notion that we can and should "fight", as a conscious effort. Then third, I'll try to explain some of the real fights that we the terminally sick do have.

So take this easy statement: "you must fight, Pieter. Don't let the cancer win!" It wraps up so many difficult emotions in a neat package. It fits into the "disease is mostly in the mind" 1970's era fantasy that still imagines meditation and positive thinking as the cure for rampaging gene mutations. And presumably cholera, malaria, and broken legs as well.

Worse is the implication of blame. When we die, did we not fight hard enough? If it takes me six months to die, am I doing a better job of "fighting my cancer" than someone who dies in six weeks? It goes beyond senseless into the cruel. We don't "lose the fight" against our cancers, any more than a cell phone loses its "fight" against battery exhaustion. The mutations will always win unless something beats them to it. It is a matter of when, not if.

That fist-pumping "you can beat it!" motif has more insidious effects. It drops responsibility like a ripening melon into the lap of the ill. It leaves the pep talker buoyed with their display of positivity and helpfulness. As a conversation with the dying, it is cheap and unintentionally nasty.

My neighbor, nice guy, every time we met over the last months, did the cheerleader thing. Finally I put on proper cancer face (shaved my head) and met him with my oxygen container, on the street outside our house. "I'm dying now, Hussein," I told him. "The treatment stopped working." He finally nodded, accepting it. Now finally we can talk about real things, like what will happen to my kids when I'm dead, and so on.

Clearing the table of the elephant poop of positivism, we see other creatures skulking about too. Worth mentioning:

- Diet positivism, from ketogenic to fresh fruit. Obviously, when you can, you should eat moderately and avoid junk foods, especially sugar. Yet a cancer patient is struggling with much more basic problems. When I have chemotherapy, I literally lose my appetite for days. Even getting a sweet pastry down my gullet can be a major victory. Thank heaven for drugs like Medrol (a glucocorticoid) that give me hunger again. From then, my body decides what it can stomach. Maybe it's buttermilk. Maybe it's chicken jalfrezi and chana masala. But heaven help you if you come to my bedside and propose that I should be eating more fruit.
- Alternative medicines and treatments, including marijuana, gene therapies, and so on. Apart from putting the responsibility for "trying hard enough to get better" onto the patient, it's poor advice. I assume you live, like me, in a city with a functioning medical system. With experienced oncologists who see hundreds, thousands of cases in their career. Who have access to databases of studies and data. With almost unlimited access to the necessary medicines. If you don't have these, then you are fairly doomed in any case. Can a single individual patient second-guess the medical machine? Is that really their duty?

Now I'll come to the real struggles of the dying. This isn't a full list, I've not done much field research. More of a sampler to show the point.

- *Finishing the paperwork.* By this, I mean cleaning up enough so that the survivors aren't punished. I've been lucky to have had time to do most of this. It's a huge job. I had hundreds of open projects and accounts. Each needs to be handed over to someone I can trust, shut down, or abandoned.
- *Fighting off the wolves.* You'd be shocked, yet I'm involved in arguments over money, and my humble yet non-zero estate. There are people who treat the dying as easy prey. I don't take it personally, instead I get my lawyers and notaries busy. I am grateful for portable oxygen and Uber, which has kept me mobile for the last weeks.

- *Managing the symptoms.* Cancer hurts. The right side of my chest aches, up to my shoulder and neck. I take opiates, two 12-hour Oxycodones, and then instant Oxonorm for moments when it's worse. With good timing I can get a night's sleep. If I mistime it I wake at 3am, from the pain. Yet I want to feel some pain, it's vital data.
- *Finding food to eat.* All my life I've been the one who shopped, cooked, served others. It makes me sad that I can't do this for my kids anymore. Yet it's pushed them to take over. My daughter does the shopping, and helps make food for me. I don't need to explain much, we know each other. I order an Indian takeaway. She prepares a thali-style plate, adds hot sauce, nukes it, brings it with a large glass of buttermilk.
- *Staying happy.* And, helping those around me to stay happy. It's far easier to be in a bad mood, tired, grumpy. People will excuse that. The chemo changes our personality, right? Well, perhaps, yet this is definitely within my control. I wrote in the Psychopath Code about emotional control. It takes effort, yet it repays itself many times.
- *Staying fit and clean.* If you've ever been bedridden, you'll know how hard it is to get up, and do things like wash your face. It's especially tough when you're getting chemo drugs that debilitate you. Yet you are only as strong as the work you do. I force myself to sit up, walk around, even to go outside if I can.

What's interesting to me is that in these struggles, other people are key. These aren't solitary conflicts. I've found that they bring my friends and family close to me. We're all involved in this slow process of dying. It may seem horrible, from some points of view. And yet, it is deeply satisfying in other ways. It has become an enriching thing, a collective work.

I'd much rather not die, yet if I'm going to (and it does seem inevitable now), this is how I'd want it to happen. Not fighting the cancer, with hope and positive thinking, rather by fighting the negativity of death, with small positive steps, and together, rather than alone.

Chapter 9. The Life FAQ

Who's the purpose of Life? Who's in charge? How much does it cost, and what colors does it come in? Today, I take all these and other Frequently Asked Questions about Life and answer them in a single easy-to-hate blog post.

Q: What's Life?

Life is the absence of not-Life, much like Beer is the absence of everything except Beer. Put it another way, Life is exactly like a huge planet-wide party involving billions of people and trillions of other life forms, lasting so many years we needed to invent "zero" just to count them, and not just playing the best music and eating the best food, but making it up on the spot.

Q: What's the purpose of Life and why should I care?

There are several competing theories about this. One popular theory is that you'd better care, otherwise raging demon spirits from an alternate dimension will descend upon you and rip you to shreds with their talons and fangs, whilst simultaneously giving you the power to heal immediately, so that the torture lasts a million years. A less dramatic theory is that Life has no purpose, and whether you care or not doesn't matter a bit, as Life doesn't give a demon's arse either way.

Q: What are the alternatives? Why is Life better?

Again, there are many theories about this, and some of the most durable and profitable ones -- also called "Religions" -- have developed multiple values of "Life". A less profitable though more scientific view is that Life is binary. You either have it, or you do not. However, Religion eats cake and drinks wine, while Science chews dry crusts.

Q: This Life sounds great, how do I get one?

Unfortunately, Life is not available for download. You can only get a Life by invitation from two existing Life owners. Also, you cannot reject the invitation. By the time you realize you have received a Life, it is too late to back out.

Q: How much does a Life cost?

For something that everyone needs and is willing to pay anything for, the surprising answer is that life is free, technically, which is the best kind of free. Some have argued that the price of a Life is a Life, though these are ironically those who usually value Life the lowest.

Q: What colors does it come in?

Frankly, mostly a shitty brown color, with periods of blue and bright red. However your Life will change colors depending on how you connect it to other Lives. Your Life's current color is often the best sign that you have connected it to the wrong, or the right people.

Q: Who invented Life? Who's in Charge?

Some people claim that Life was invented by one or more space ghosts who rule the Universe with extraordinary jealousy, paranoia, and Machiavellian cunning. In this story, when things go right it's always thanks to the space ghosts, and when things go wrong, it's always your own fault. The space ghosts of history have no emotions except violent anger and suffocating love, are never wrong, and never say sorry. Happily, space ghosts are not real. We can safely state that no-one invented Life and no-one is in Charge. As I said, Life is a big party.

Q: How can I make money from Life? What's the business plan?

You can make good money by projecting your charismatic psychopathic inner-self onto others in the form of space ghosts, or similar hoaxes, if you are that kind of algorithm. The alternative algorithm is to find ways to make your Life useful to others, at which point you can reap a little of that benefit.

Q: What's the license for Life? Is it open source?

Life uses a viral no-attribution share-alike license. No-one has successfully written down the Life license yet, so we can't ask the Open Source Initiative whether it's a compatible license.

Q: How do I boot up my Life?

Most Lives come with convenient Parent plugins (primary, and backup), which will run your Lives for you until you are just getting cocky enough to believe things can't crash. After 18 you can choose to borrow money to try to upgrade your Life, or go rent your Life to the Man in exchange for food and shelter.

Q: I don't like my Life, who can I complain to?

You will find good company with the 75% of others who dislike their Lives too. Usually, if you let them complain to you, they'll tolerate your complaining back. Mutual complaining is the basis of much social interaction. Note however that no matter how much you complain, no-one else can fix your life except you.

Q: My Life is slow, and crashing, how do I fix it?

That's an awfully good question. There is no single answer. It is true that Life does randomly crash, and we just have to get used to that. Generally poor performance may be easier to work on. You may be connected to other Lives in dysfunctional patterns -- if you can identify these, change them. Your hardware may have issues, especially if you are an older generation model, or you abuse your co-processors. Life does also catch viruses and other malware. Consider a regular checkup.

Q: My Life is lousy. Can I get an exchange or refund?

No, this is not possible for technical reasons we can't explain here. Just trust us. And if anyone offers you a refund or exchange, watch out, they are crooks. They just want your money or your Life. In Life, if something or someone appears to be too good to be true, they inevitably are.

Q: I sat on my Life and bent it. How do I fix that?

Above all, if your Life still works, leave it alone. Trying to fix a Life that isn't broken will often make it worse. Some Lives are better bent than straight.

Q: Why is Life so unreliable? It just doesn't work for me!

This is just a FAQ, not a full guide. On the one hand, you are the proud owner of a unique Life, yours from the day you are born to the day you die. On the other hand, there are no instructions. You can learn a lot from watching other people mess up their Lives. However the very best way to make your Life more reliable and overall, enjoyable, is to mess up yourself and then recover quickly.

Q: I find Life kind of amateurish. Is there an Enterprise version?

You can buy lots of upgrades, and some of these do make a difference. However ultimately your Life has a fixed period of validity and there is nothing you can do about that. The best way to get free upgrades for your Life is to ask nicely. You can also be a sociopath, and that works for some people, especially those who want Enterprise Life.

Q: Can I get someone to help me understand my Life?

Again, this is but a humble FAQ and a profound understanding of Life is typically something you will get only when it's tragically, and irreparably late. Beware anyone claiming they can help you with your Life, as the vast majority of people barely have their own Lives under control.

Q: How do I contribute to Life? Where's the source code?

The Life community welcomes all contributions. The standard approach is to find a second contributor of the opposite gender, and then use the fork and merge model, repeating as necessary.

Q: Without strong Intellectual Property Rights, surely no-one will invest in new life!

Oh, just go away!

Q: What hardware do I need to run my Life on?

Life is entirely self-hosting, so you cannot switch the hardware out for a different box. Make the most of the hardware you have. Though you can extend it, and some people make a good job of changing it, it is usually a bad idea to make changes once you're up and running.

Q: If I buy larger hardware, will it make my Life better?

You will get more stares, though that will not make your Life run better, and indeed will usually mess it up permanently.

Q: My life and I always argue. What's the answer?

Do you mean "wife"? Then stop trying to always be right. 42.

Postface

I hope you enjoyed this.

I'd like to dedicated this book to the people who took care of me in my last weeks and days.

To my sister Helen, who came from far to cook for me and stuff our fridge with food. To my mother Sheila who spent so much time in Belgium we wondered if she was going to emigrate from France. To my son Freeman who made me the best omelets. To Noemie who with the teenager's sigh made me endless hot milks, coffees, teas, and dishes to eat. To my sisters Ann and Kristien who could not often be with me, yet offered their love and support by distant phone. To Gregor who came and saw me in pain and gave me a long, silent hug. "I love you, Daddy," he said, in that embrace.

To the nurses and doctors of Unit 32 in Brugmann who I cannot cite by name, yet who hovered like protective angels over me twenty-four hours a day, catering literally to my every need.

To the nurses and doctors of the Palliative Unit in Brugmann, where I'm moving to next week.

Brussels 30 September, 2016