Mentofacturing

Vincent Lextrait

vincent@lextrait.com
http://www.mentofacturing.com

October 27, 2025

Acknowledgments

The author wishes to thank Denis Arnaud, Christophe Bousquet, Dietmar Fauser, Ion Petrescu, David Wales and especially Edwige Gladwin who have been kind enough to read the two first chapters of this document, some of them repeated times, and to suggest invaluable additions and corrections.

Thank you also to Yves Djorno and Olivier Richaud for reviewing the third chapter.

Their help does not constitute an endorsement of this work.

Contents

List of Figures

Chapter 1

From Manufacturing to Mentofacturing

We are like dwarfs sitting upon the shoulders of giants.
We are able to see more than the ancients, and farther,
not because of the acuteness of our sight, or our own height,
but because they bring us up and raise us with all their gigantic elevation.

Scholar Bernard de Chartres, circa 1124-1130

1.1 Introduction

In 1968, Douglas Engelbart, before personal computing existed, was talking about the "intellectual worker", referring to people using a computer "alive for you all day". In 1970, Alvin Toffler [57, 58] baptized the post industrial period the "Information Age". In 1991, in the introduction of an article entitled "Mentofacturing: a vision for American industrial excellence" [18], Gordon E. Forward, Dennis E. Beach, David A. Gray, and James Campbell Quick wrote that "Mentofacturing" companies, focusing on making with the mind rather than making with the

hand, would look more like software companies. In 2001, Peter Denning and Robert Dunham [14], explained that the information technology profession (usually shortened as "IT") was the first profession of the third wave of civilization. Because of all this, studying the software industry is of paramount importance to understand the Information Age itself, how people will interact, where the organization of work will find its new rules, and in which way work and our lives themselves will be redefined. Indeed, the pioneering software industry in the Information Age will play the same role as the pin-making factory that Adam Smith studied at the dawn of the Industrial Age, in 1776 [49], with the propagation of its new practices to all industries and to society itself.

Surprisingly enough, after more than half a century, and in spite of its impact on world economy, or the *wealth of nations* to paraphrase Smith, the software industry is still very imperfectly understood. No document of the same nature as Adam Smith's case study of a pin-making shop exists, and a lot, if not all knowledge, remains purely empirical. The mechanisms governing productivity were at the root of Smith's work, and they are still the focal point of interest. For instance, experimental recipes to organize software production in a more efficient manner are regularly proposed. These heuristics work to some level, but to reach higher and possibly optimal productivity, in the same fashion as the pin-making activity case study, it is necessary to understand the fundamental laws of software production. Only then the key factors for productivity can be exploited.

Some specific subjects have been investigated, such as the influence of office setup on productivity. Tom DeMarco and Timothy Lister show in "Peopleware" [13] that the ratio of productivity between developers located in regular offices versus cubicles reaches a factor 2.6 in favor of regular offices. David Thielen, in 1991 [55], already insisted on individual offices, and reiterated this in 1999 in his book on Microsoft practices [56]. Individual offices aside, the 2.6 ratio between regular offices versus cubicles, is already a stunning result. As we'll see, a lot more parameters influence far more deeply developers productivity.

Given the ever growing importance of software production in the industry, its position as the asymptotic and incompressible residue of automation, given the deep impact it has on all modern human activities, it is striking to note that, in contrast, nobody has tried to analyze it in depth. The lack of publications on the subject explains in part the very eclectic nature of this document's bibliography.

To a large extent, software production is still based on early and seminal works on manufacturing industries, which were carefully studied by Adam Smith [49], Charles Babbage [3], Karl Marx [34], Frederick Taylor [53, 54], Alfred Sloan [48], and many others. Now, who would agree today that Software Production has anything to do with manufacturing? Certainly nobody.

In spite of that, the organization of people producing software is still deeply affected, largely unconsciously, by the same old principles. Software production is still relying on Adam Smith's analysis of the "Pin-Making" case and his recommendation on the *division of labor*. This idea, not completely original, was presented more than two centuries ago, in 1776 and further analyzed by Babbage during the first half of the 19^{th} century. A very long time ago.

We are going to show that Adam Smith himself knew that the division of labor was a temporary step towards a new age, which we know now as the "Information Age". This work attempts to illustrate the inadequate influence of Adam Smith's proposal on software production, an influence Adam Smith himself would not have supported. It tries to explain the reasons of this inadequacy, which were already envisioned by Adam Smith. It intends to shed some light on the software production process and to show some little, or close to never discussed productivity-related parameters. It also gives a number of hints on how to drastically enhance productivity through adapted recruitment and organization. Lastly, it elaborates on why the fundamental laws of software production are, more or less intentionally, simply ignored.

In a nutshell, this document is presumptuous enough to try proving both Frederick Brooks and Capers Jones wrong. Indeed,

in his 1995 revision of his 1975 best seller "The Mythical Man-Month" [7], Brooks postulated that no silver bullet existed for enhancing developers' productivity, and that there was no single strategy, technique or trick that would exponentially raise the productivity of programmers. In "Assessment and Control of Software Risks" [9], Capers Jones stigmatized the lack of specialization as one of the risk factors endangering software projects. This document plans to demonstrate that it is on the contrary, its biggest threat.

It will become clear that the content of this document is not only relevant to software development, but to human intellectual endeavours involving creativity, innovation, a variety of software tools and large teams. Movie and music making fall into this category for instance. Philosophically, the framework which is closest to this work is Michael Hammer's "Business Process Reengineering" [22, 23].

1.2 The division of labor originates from manufacturing

The division of labor, or labor parcellization, sometimes also referred to as "job stratification", or simply "specialization", was born as early as the neolithic age (in western Europe, more precisely the Magdalenian Age, which belongs to the Upper Paleolithic, between c. 18,000 and 10,000 BP). Indeed, historians consider that specialization started to exist when sedentarization, and intensive trade began. At that time, our ancestors had more than one hundred different tools. Among these tools, there were various kinds of flint stones meant for different purposes. Flint stones were not excavated or discovered, shaped and polished by the same people. Each of these activities was endorsed by different specialized groups, leading to commerce of goods over large distances.

Throughout history, the division of labor was successively studied by Plato (428/427 BC - 348/347 BC), Xenophon (427-355 BC), William Petty (1623-1687), Bernard de Mandeville

(1670-1733), and David Hume (1711-1776). However, true industrial specialization, meant to cut mass production costs drastically, is an idea ordinarily attributed to Adam Smith (1723-1790).

Historical truth is somewhat different, as explained by Murray Rothbard in "An Austrian Perspective on the History of Economic Thought", Rothbard [46] shows that Adam Smith actually plagiarised somebody else, and borrowed the idea of the division of labor, and even the Pin-Making case study, from a former source: Henri-Louis Duhamel du Monceau (1700-1782). In his introduction to René-Antoine Ferchault de Réaumur's l'"Art de l'Épinglier" [44] - The Art of the Pin-Maker -, [44] Duhamel du Monceau wrote:

> There is nobody who is not surprised of the small price of pins; but we shall be even more surprised, when we know how many different operations, most of them very delicate, are mandatory to make a good pin. We are going to go through these operations in a few words to stimulate the curiosity to know their detail; this enumeration will supply as many articles which will make the *division of this work*. [...] The first operation is to have brass go through the drawing plate to calibrate it. [...]

As a matter of fact, it seems [39] that the expression *division of [this] work* (*division de [ce] travail*) in du Monceau refers to the subdivisions of the *text* which follows, not the *division of labor*. Smith had travelled to France, and spoke French, but even for the native French speaker who would superficially look at the text the ambiguity could easily lead to misinterpretation. It is funny to realize that most likely, the expression *division of labor* stemmed from a mistake.

Equally funny, but also enlightening, is the fact that René-Antoine Ferchault de Réaumur, who wrote "The Art of the Pin-Maker", describing different trades with drastically different roles for individuals, was among other activities as a scientist, a famous entomologist [17], very familiar with taxonomy.

ART
DE L'ÉPINGLIER. (1)

INTRODUCTION ; par M. DUHAMEL (*).

1. ON ſait en général, que les épingles ſont des bouts de fil de métal, pointus par un bout, & garnis d'une tête à l'autre bout, & que leur uſage eſt d'attacher de la toile ou d'autres étoffes ſans les endommager, de façon qu'on puiſſe ſur-le-champ, & en tirant l'épingle, déployer l'étoffe & la rattacher de nouveau : les femmes en font une grande conſommation, ſur-tout pour leurs coëffures.

2. IL n'y a perſonne qui ne ſoit étonné du bas prix des épingles ; mais la ſurpriſe augmentera ſans doute quand on ſaura combien de différentes opérations, la plupart fort délicates, ſont indiſpenſablement néceſſaires pour faire une bonne épingle. Nous allons parcourir en peu de mots ces opérations, pour faire naître l'envie d'en connaître les détails ; cette énumération nous fournira autant d'articles qui feront la diviſion de ce travail.

3. 1°. COMME le fil de laiton ſe vend en botte aux épingliers, il ſe trouve

(1) L'art de l'épinglier eſt un des premiers qui ait été donné par l'académie. Il fait partie du premier volume de la traduction allemande, publiée en 1762, avec les notes de M. de Juſti.

(2) On n'a trouvé dans le dépôt de l'académie, qu'un ſimple projet de mémoire ſur l'*épinglier*, fait par M. de Reaumur, & trois planches gravées, mais ſans lettres de renvoi & ſans explication des figures. M. Duhamel, qui s'eſt chargé de mettre ce mémoire en état d'être imprimé, y a fait encore graver trois nouvelles planches (2). M. Perronet a fourni auſſi pluſieurs articles qui enrichiſſent cette deſcription. On a cru devoir indiquer ce qui appartient à chacun de ces auteurs, en mettant leur nom à la fin des articles qu'ils ont fournis.

(2) Ces planches, chargées de détails inutiles, & deſſinées avec luxe, ſe trouvent ici réduites à une ſeule, ſans que cette économie nuiſe à la clarté du diſcours.

Xxx ij

Figure 1.1: L'Art de l'Épinglier (The Art of the Pin-Maker), with *"division de ce travail"* highlighted

When you see his insects plates, you are overwhelmed by the idea of diversity and specialization, including within a species (see figure 1.3, page 16). The presence of different specialized individuals, each with a modest contribution to a greater goal is a striking similarity between a wasp hive and a pin-making fac-

tory implementing specialization of work. No surprise Réaumur became interested in labor parcellization.

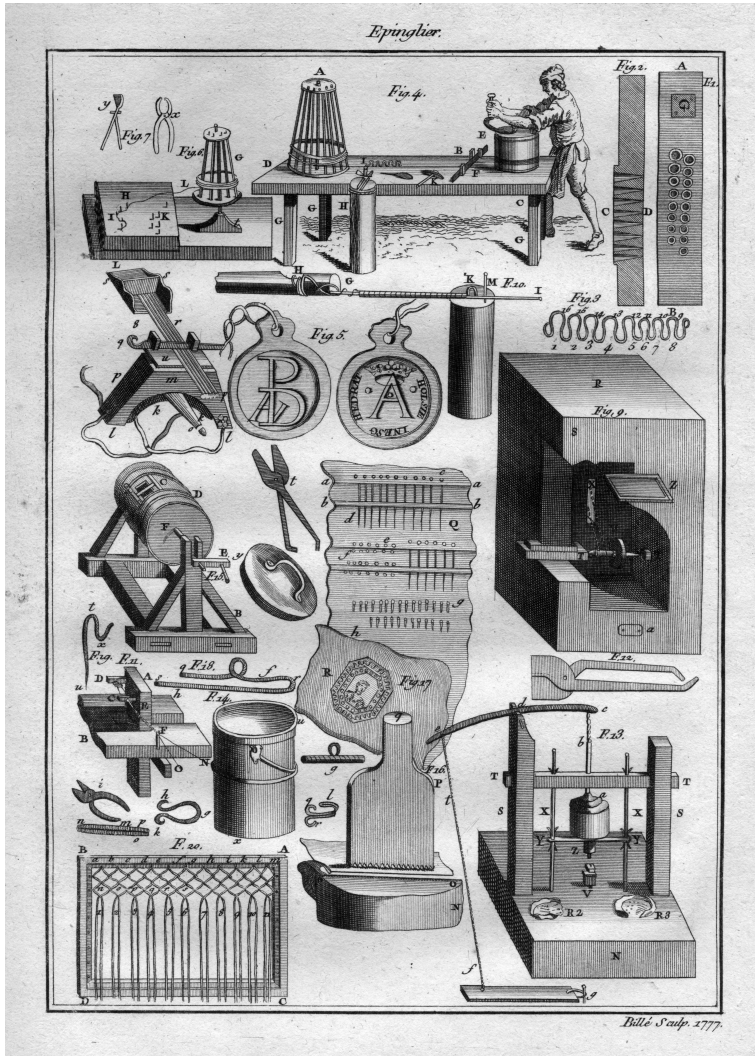

Figure 1.2: A plate showing different types of tools to produce pins, in Ferchault de Réaumur's *"L'Art de l'Épinglier"*, *"The Art of the Pin-Maker"*

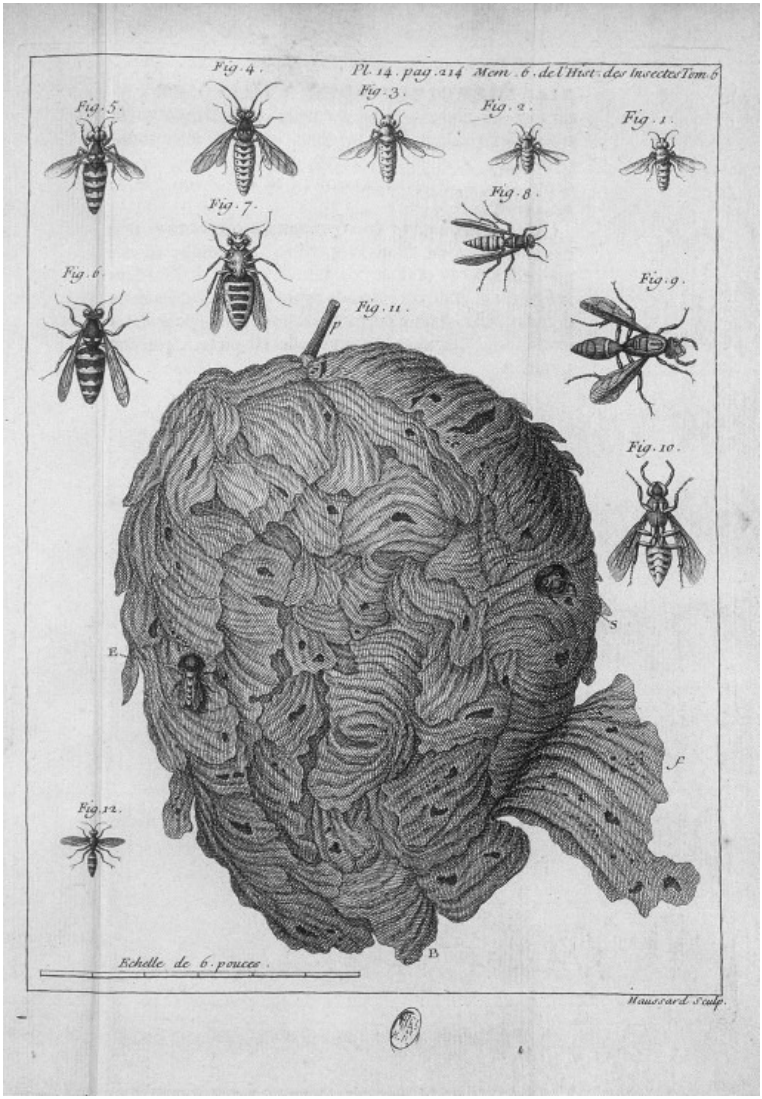

Figure 1.3: A plate on different types of wasps, in Ferchault de Réaumur's *"Mémoires pour servir à l'Histoire des Insectes"*

So, Smith was not the original author of the ideas attributed to him, but it is important to note that, as an author and a

clubber, he powerfully popularized them. These ideas deeply influenced *manufacturing*. They were even *tied* to manufacturing. Babbage, in 1832, entitled his followup work "On the Economy of Machinery and Manufactures". Neither Smith nor Babbage intended to propose the division of labor for non manufacturing activities.

The first one to deviate from this goal was Frederick Taylor [54] in 1911. Taylor wrote:

> It is hoped, however, that it will be clear [...] that the same principles can be applied with equal force to all social activities: to the management of our homes; the management of our farms; the management of the business of our tradesmen, large and small; of our churches, our philanthropic institutions, our universities, and our governmental departments.

This is, to say the least, optimistic. It can be argued whether Taylor intended to talk about the division of labor. Actually, Taylor has never made a single reference to the division of labor in his works, although in public eyes, especially European ones, he is often mistaken with Adam Smith as the father of the parcellization of work. However, Taylor repeatedly mentions the existence of *trades*, and the division of labor goes without saying for him.

In order to understand why Adam Smith's proposal applies to manufacturing, we need to understand why the division of labor increases productivity. Adam Smith explains it: when a single person is responsible for all the production steps, like in craft industries, his work involves many different activities, requiring different tools or machines. Switching from one tool or machine to another one takes time, and introduces cost, because of physical movement from one spot to another one, or merely because of the change of tool. This cost grows linearly as a function of the number of activities.

Instead of endorsing successively different trades, a workman should concentrate on a given trade, and labor should be divided to specialize people. By doing so, the division of labor allows

to eliminate the hidden linear cost. The more activities are involved, the more labor should be divided, and the more this new organization allows for savings. Adam Smith showed how to move away from craftsmanship and reach what has been known as an "industrial" organization.

In that explanation lies the fact that the division of labor should not apply to all human activities. We saw that the division of labor attempts to resolve the cost related to switching from one tool or machine to another one. It should not be applied to human activities which do not involve different tools or machines, or which do not imply a meaningful cost when switching between them.

As we shall see, although Taylor's statement of universality is amusing *a posteriori*, his overly enthusiastic point of view is, surprisingly, still shared by many people organizing software production. Today, unfortunately, very few people in the software industry have actually read Adam Smith or know the root motivation for the division of labor.

Given the clear insight Adam Smith had about manufacturing, did he think that the division of labor was a final solution for the organization of work? As a matter of fact, he didn't, and it is clear in the Wealth of Nations.

1.3 The division of labor is a self destructive phase

Not only did Adam Smith propose the division of labor solely for manufacturing, he also explained that was only a transient phase allowing for the start of the revolutionary rise of automation. Adam Smith himself was therefore fully aware of the self destructive nature of the division of labor. He explains it the following way:

> [...] every body must be sensible how much labor is facilitated and abridged by the application of proper machinery. [...] I shall only observe, therefore, that the invention of all those machines by which labor

> is so much facilitated and abridged, seems to have been originally owing to the division of labor. [...] in consequence of the division of labor, the whole of every man's attention comes naturally to be directed towards some one very simple object. It is naturally to be expected, therefore, that some one or other of those who are employed in each particular branch of labor should soon find out easier and readier methods of performing their own particular work, whenever the nature of it admits of such improvement. A great part of the machines made use of in those manufactures in which labor is most subdivided, were originally the invention of common workmen, who, being each of them employed in some very simple operation, naturally turned their thoughts towards finding out easier and readier methods of performing it. Whoever has been much accustomed to visit such manufactures, must frequently have been shewn very pretty machines, which were the inventions of such workmen, in order to facilitate and quicken their own particular part of the work.

Adam Smith saw what we call now pompously "offpeopling"[1], very clearly, as early as during the mid-eighteenth century. Machinery would ultimately replace many different trades, and the division of labor was only an initial phase towards ultimate automation.

Here is an interesting anecdote he quotes:

> In the first fire-engines[2], a boy was constantly employed to open and shut alternately the communication between the boiler and the cylinder, according as the piston either ascended or descended. One of those boys, who loved to play with his companions,

[1] One of the historical steps in complete offpeopling was reached in 1987, when Steve Jobs finished the construction of a fully automated factory for the first product of NeXT Computer, the NeXTcube.

[2] This was then the designation for steam engines.

> observed that, by tying a string from the handle of the valve which opened this communication to another part of the machine, the valve would open and shut without his assistance, and leave him at liberty to divert himself with his play-fellows. One of the greatest improvements that has been made upon this machine, since it was first invented, was in this manner the discovery of a boy who wanted to save his own labor.

Adam Smith indicates another source of automation breakthroughs, the makers of the machines themselves:

> All the improvements in machinery, however, have by no means been the inventions of those who had occasion to use the machines. Many improvements have been made by the ingenuity of the makers of the machines, when to make them became the business of a peculiar trade [...]

It means that Adam Smith clearly envisioned the transient Industrial Age and the situation beyond. Indeed, as Alvin Toffler [57, 58] explained it, after the Agrarian Age, followed by the Industrial Age, the Information Age arose. It is interesting to remark that the Industrial Age, based on the division of labor, actually *gave birth* to the Information Age. Without underestimating the genius of the Information Age fathers, all the following events belong to a philosophical continuum initiated, and to some extent, envisioned, by Adam Smith:

- 1776: Adam Smith's recommendation for maximal division of labor;
- 1830-1842: Charles Babbage studies Adam Smith works on the division of labor, and creates his Analytical Engine;
- 1880: Herman Hollerith's tabulator;

- 1934: The atomic operations — ultimate parcellization of processes — proposed by Alan Turing in his universal machine;
- 1936: Konrad Zuse Z1 computer;
- 1937-1942: John Vincent Atanasoff and Clifford Berry's digital computer;
- 1943-1946: John William Mauchly and John Presper Eckert's ENIAC;
- 1947: John von Neumann's machine;
- 1970: the Intel 4004 processor;

The transition between the pre and post-Turing eras may sound like a discontinuity, even if it is obvious that the Turing Machine attempts to reach parcellization quanta — the instructions —, which remind of Adam Smith's trades processes parcellization. There are additional links between the mechanical progress done until Turing and the apparently abstract concept he coined in, as a mathematician. The term used by Turing to describe his concept was "machine", and, this is less widely known, Alan Turing, with his machine, had the intention to describe mathematically "mechanical processes". This expression, widely forgotten now, was replaced by the nineteenth century acceptation of the word "algorithm", which obliterated the links with the Industrial Age, leaving only the word "machine" in Turing contribution, as a memory of the old age.

Since the Wealth of Nations, automation has been a path of no return (but anecdotic) to the division of labor. While automation used to create — more or less — brutal discontinuities during the Industrial Age, the Information Age introduces a neverending disruption, a *continuous revolution* process. Software is now involved in every of our familiar objects. Furthermore, software is *reflexive*, it automates its own production, and its production is a continuous innovation process. Brutal technological discontinuities exist, but within a constant revolution. In

order to convince yourself of the relationship between the Information Age and revolution, simply make a search on Amazon on books related to IT and containing the word "revolution" in the title, at the time of writing, you would get around 250 hits. This is not accidental, it tells a lot. Grace Murray Hopper, one of the very first programmers, insisted untiringly that managers shouldn't be afraid of change. In her opinion, "the most damaging phrase in the language is 'We've always done it this way.'".

The term revolution is not accidental, as it found a powerful echo during the counterculture period of the 60's. Stewart Brand [6] explains it clearly enough in his paper for Time Magazine: "We owe it all to the hippies / Forget antiwar protests, Woodstock, even long hair. The real legacy of the sixties generation is the computer revolution". Even to date, some of the inheritants of the hippies are still working in IT, Richard Stallman being one of the icons in the category. Steven Levy [33], in "Hackers: heroes of the computer revolution" explains it very clearly.

Since the Information Age implements a continuous revolution through software production, is Adam Smith's division of labor, meant for the Industrial Age only, a reasonable organization for software production?

1.4 Software production and the assembly line

Considering software production as an assembly line, or specifically as a factory, an auto-proclaimed reminiscence of manufacturing, has openly and repeatedly been done. "Software Factory" is a very unfortunate expression, because it is very misleading. It conforts people into thinking that software production and manufacturing have something in common. The

venerable Bob Bemer, inventor of time sharing[3] and ASCII[4] introduced it. In his mind it meant a factory *constituted* of software. Unfortunately, the insistance of many to build a factory *for* software, inspired from the division of labor, with human beings, has prevailed.

The love or sheer hatred that some professionals have for or against software factories is clearly described by Ivan Aen, Peter Bøtcher and Lars Mathiassen in "The Software Factory: Contributions and Illusions" [1]:

> The term factory signals a commitment to long-term, integrated efforts — above the level of individual projects — to enhance software operations. This is not only a powerful, but also a necessary idea taking the challenges involved in professionalizing software operations into account. But for many the term factory has at the same time the controversial connotation that software development and maintenance is comparable to mass-production of industrial products, and arguably this is not the case. This can easily lead to illusions with respect to the kinds of interventions that can, in fact, improve software operations. It is not surprising, therefore, that some software professionals like the concept while others do not.

One of the very famous initiatives of the past is one of EUREKA[5] avatars: EUREKA Software Factory. ESF started as a 3,000 man-years project founded by a group of European partners in 1987, with the intent to improve the process of large-scale

[3] A technique allowing several users to interact with a computer in the same time, instead of queueing, as it was the case before its invention. Time sharing allocates a slice of time of the computer to each of the users, in rosters.

[4] American Standard Code for Information Interchange, a code now universally used to encode characters into 8 bits numbers.

[5] EUREKA used to be a pan European industrial project aiming at boosting Europe's competitivity. It started in 1985.

software production. ESF was supposed to deliver not only technical solutions, but also a model of organization. It reached its end in 1997, short of any remarkable impact on the day to day life of European software development organizations. The factory model did not bring a valuable contribution to software production.

Most people working in the software business hate to look at the past. This must be related to the intrinsic revolutionary nature of IT, and the fact that professionals eagerly look at what lies in the future, while being heartless for the past.

If we try to take a historical standpoint on the history of software development, we notice that the number of trades involved in software construction was from the beginning already quite large. There were architects, analysts, keyboarders, operators, punch operators, programmers, maintenance personnel. But it never was comparable to Adam Smith's report on pin-making shops:

> One man draws out the wire, another straights it, a third cuts it, a fourth points it, a fifth grinds it at the top for receiving, the head; to make the head requires two or three distinct operations; to put it on is a peculiar business, to whiten the pins is another; it is even a trade by itself to put them into the paper; and the important business of making a pin is, in this manner, divided into about eighteen distinct operations, which, in some manufactories, are all performed by distinct hands [...]

Nothing to do either with Frederick Taylor's estimate on regular industrial establishments:

> In an industrial establishment which employs say from 500 to 1000 workmen, there will be found in many cases at least twenty to thirty different trades.

The number of trades in the software industry, with time, has deeply evolved. Some have disappeared, thanks to the reflexivity of software helping to automate software construction

tasks themselves. Some have been blooming. Many if not most organizations today have specialists for various aspects of software production. Some organizations even established separate groups for software construction and software maintenance, two different trades being used for those two activities. Common sense says that if such division of labor is implemented, production costs are high, and the construction trade looses any feeling of responsibility. Only failure lies as at the end of the road.

The unabated obsession to create trades can clearly be attested by the type and variety of software development job advertisements published in the press. A specialist of such and such tool is required, sometimes with a specified version number. This reminds unavoidably of manufacturing. The people publishing these ads are expecting to fill *trades* operating in a *factory*.

But what are the main mechanisms pushing towards considering software production as analogous to manufacturing, and attempting at all costs to apply the division of labor?

1.5 The over sensitivity about salaries

From a somewhat simplistic standpoint, the software world is split between *professionals* and *user* companies. Among those user companies, you find Banks, Airlines, Insurances for instance. Being a *user* company does not mean that the company is not doing software development. It means that the main mission of the company lies somewhere else. This is very clear for the business areas listed above. Most of the large *user* corporations have actually very large IT departments. It is not uncommon to find thousands of people in the IT department of a user company.

In a user company, IT is perceived a cost because there are, unfortunately, very little possibilities to measure the actual savings IT allows to make. When a project is initiated, very scarce are the situations where business processes are kept identical, and the avid expectations everybody has on the level of automation usually exceed by far existing practices. As two situations,

equivalent in terms of processes, cannot be compared, the added value IT brings cannot easily be assessed.

Cost being the obsession of the IT department, it is very natural to keep a very tight control on all the elements of the cost structure. After all, the cost in an IT department is rather simple to analyze: the cost deriving from the consumption of external resources is in the hands of purchase professionals who do their job at negotiating prices very well. The only meaningful residual *variable* cost is the salary mass. As a natural consequence, in order to keep the IT cost effective, the natural conclusion is to keep salaries under tight control, and as low as reasonably possible. Statistically, this does not allow for recruiting the top tier individuals, rather the bottom one. We must remember that Taylor, in 1911, even in manufacturing industries, pushed for avoiding minimal salaries, and pleaded for higher ones allowing for even higher productivity. The idea of minimizing salaries is therefore going even against classical manufacturing rules.

By trying to minimize salaries, most users limit the analysis to only one parameter of the actual cost, the individual cost *per unit of time*, in other words, the salary.

William Waddell and Norman Bodek, in "Rebirth of American Industry - A Study on Lean Management"[59], have highlighted the dangers of such an approach, inspired from Alfred Sloan, who invented "management in itself" as a direct inheritance of the idea of specialization (why bothering about technology, there are specialists for that). Their book is about manufacturing, not on software. In order to explain how silly it is to consider staff as a "variable cost", they try to take the most obvious example they can take:

> To understand the magnitude of Sloan and DuPont's debacle, the best analogy today would be to look again at one of the companies with the funny names. Imagine Google, Yahoo or Microsoft declaring that computer science is a commodity — basically any warm body from the local temp agency can do it —

> and that the key to success in running these technology companies is not technology, but finance and marketing. Imagine further that they all but declare war on their programmers and system design folks, classifying them as variable costs and devising a management system aimed directly at cutting their numbers and minimizing their pay.
>
> The only way such a scenario would allow those companies to survive with such a hair-brained strategy would be (1) if all of the other American tech companies did the same thing and (2) another world war broke out and the Air Force blew up every other computer in the world.

Unfortunately, this situation exists frequently in technology companies, especially the ones which have gone vertical.

So, what should companies do if staff is not a variable cost?

What really matters is not the salary, it is the unitary cost C for an employee to achieve one unit of production. Very plainly, $C = St$, where S is the cost of the employee per unit of time (the salary), and where t is the time required to deliver one unit of production. This product is the actual cost. This cost is the one which needs to be minimized, not simply the S factor. Considering only the S factor leads to minimizing individual salaries. Considering only the variations of the S factor, in other words, means neglecting the variations of the t factor. What does it represent? The t factor measures the time the individual requires to produce one unit. This is the measure of productivity. Neglecting its variations over variations of S means that interpersonal variations in productivity are negligible over salary variations. Is it that obviously true? At least, it can be perceived true, as the literature available for decision makers is mostly silent about interpersonal productivity variations. Only very stubborn and systematic readers can find a few scattered tracks of it.

Before proceeding, a word is required about the mysterious "unit of production" used above. It can be debated whether a unit of production exists in software. It is well known that

immediate ideas that come to mind to define it are inadequate. A line of code for instance is not a very good unit, as we all know that verbose code can be produced very quickly. Better ways to measure the "weight" of a piece of software have been developed with time, such as Allan J. Albrecht's *function point analysis* [2]. It is still questionable, as it measures the "weight" of a *given solution* to a problem, and not the weight of the *optimal solution* to a problem. *COCOMO* (for COnstructive COst MOdel) [5] is no different. Everybody knows that there are enormous variations between the two. The optimal solution is very frequently the smallest in size (same as in mathematics by the way). Challenges existed in the early days of software development to remove a few instructions, to reach even higher performance. It was called "code bumming". This was not merely optimization, but required completely revisiting the solution and changing the algorithm. This search for an optimal solution brings, even today, interesting mathematical results.

Bjarne Stroustrup, the inventor of the C++ programming language, and Professor at the Texas A& M University, gave me his view on that specific topic:

> The best programmers apparently effortlessly produce code of a completely different level of quality at a rate that is very much higher (i.e. you get both quality and quantity — and they tend to solve each problem with far less code than the average programmer).

Knowing *a priori* the complexity to deliver a piece of software (which is the number of units of production it requires) is not only more difficult than measuring the complexity of a given solution, it is clearly impossible. Producing software is pure innovation, and innovation cannot easily be planned. Furthermore, if a method allowing to compute the complexity of a software problem *a priori* existed, it should be able to determine easily that the cost of an insoluble problem is infinite. We all know that this is impossible, it is mathematically undecidable.

All this being said, the absence of a fully satisfying "unit of production" should not prevent us from using imperfect ways to approach it. Function point analysis, COCOMO, and even number of lines of code are very interesting indicators.

Anyway, even if no unit of production exists in the software field — I am not far from believing that for the reasons listed above — the relative productivity of individuals can be evaluated on a given stated problem. The time to find a solution and the cost associated to it, for a precise given quality of delivery[6], can both be measured. As a conclusion, *relative* productivity can be defined and measured without requiring a production unit.

Let us now investigate the variations of the neglected parameter: productivity.

1.6 Interpersonal discrepancies of productivity

Even when they are aware of it, very few IT managers would publicly admit that huge productivity discrepancies exist between people in their staff. We shall discuss later why this is so. Meanwhile, it is a fact that ratios of productivity between developers reach 1 to 2 very easily, 1 to 5 frequently, 1 to 10 rather often, while 1 to 20 or more happens reasonably often. Some cases exhibit ratios of one to the infinite. Nathan Myhrvold, former Chief Scientist at Microsoft says:

> The top software developers are more productive than average software developers not by a factor of 10X or 100X, or even 1,000X, but 10,000X.

Similarly, Robert L. Glass [20] writes:

> The best programmers are up to 28 times better than the worst programmers.

[6]It is very important to set the level of quality expected, as, if it was not defined, it would be easy to degrade it to improve the time to deliver it.

Steve Jobs is known to have declared that the ratio of productivity between the best developers and the average ones reaches 100. In *The Lost Steve Jobs Tapes*, which record conversations Jobs had with journalist Brent Schlender in June 1995, we can hear:

> In most businesses, the difference between average and good is at best 2 to 1, right? Like, if you go to New York and you get the best cab driver in the city, you might get there 30% faster than with an average taxicab driver. A 2 to 1 gain would be pretty big.
>
> The difference between the best worker on computer hardware and the average may be 2 to 1, if you're lucky. With automobiles, maybe 2 to 1. But in software, it's at least 25 to 1. The difference between the average programmer and a great one is at least that.
>
> The secret of my success is that we have gone to exceptional lengths to hire the best people in the world. And when you're in a field where the dynamic range is 25 to 1, boy, does it pay off.

This is a little secret of the software industry, which is never openly debated, and which raises a great deal of uneasiness. Discrepancies of productivity can of course arise if discrepancies of deliverables quality are accepted. But even in the case apples and apples are compared, and the same level of quality is reached, ratios of productivity from one person to another can reach the factors listed above.

One to the infinite is the easiest to explain. Some developers, when confronted to a given problem are *convergent* and some others are not. What does that mean? Among the various tasks that a developer is endorsing, bug tracking is one that requires a combination of creativity and extreme rigor. Some pretty bad developers, when facing a difficult piece of software, tend statistically to introduce slightly more than one bug every time they fix one. Their activity never converges, and their

deliveries never pass qualifications. Such developers do exist. After all, at least half of the significant software projects in the world ultimately fail, and people like that can stay unnoticed during ages. When considering their productivity with respect to the one of a converging developer, the ratio of productivity is infinite.

Let me give another real life example. I had in my team a developer who had been working one year or so on a reputedly moderately difficult problem. But we know it is impossible to assess the difficulty of a problem *a priori* when it is not falling into a very standard model. After one year, the piece of code he was working on was never qualified to go to production. Every time he tried to get it through quality assurance, it would fail at some point. The developer would work on it for a while, would believe he had fixed the bugs, would submit it again, and would fail yet another time. I could have concluded:

- That the problem was intrinsically more difficult than expected, and no developer would ever be able to deliver a solution, or
- That the developer was *divergent*, and that another one should take over his task.

How to choose between those two? I chose to select one of the best developers I had, Dietmar Fauser. Somebody who had repeatedly proven his ability. If he succeeded, I had a proof the former developer was not convergent (in front of the particular task he was assigned to), if he failed, I had enough evidence that the task was very difficult.

After the replacement happened, the piece of software was delivered in one day and a half. It passed all qualifications successfully. This was many years ago, we never heard anymore about it.

Was the ratio of productivity between the two developers 365/1.5, that is in excess of 240 times? Very likely not. I believe the first one would never have delivered in a lifetime. The

first developer was not convergent, and the ratio of productivity between the two, in the context of this problem, was infinite.

Now, what about ratios from 1 to 5, one to 20, even when comparing convergent developers?

Examples of very high productivity are, of course, not recent phenomenons, and some are now part of history. The first recorded "feat" in terms of productivity was recorded at the time of the MIT pioneers. It was the one of Alan Kotok, Peter Samson, Bob Saunders, Bob Wagner and a couple of others. In 1961, they developed an entire assembler for the PDP-1 over a single week-end.

Bill Gates, before founding Microsoft, when still a student at Harvard, called MITS, the makers of the Altair 8800, to tell them he had an implementation of the Basic programming language for their machine. He had nothing, and nobody, even the most gifted, had been able to make a Basic interpreter fit into such a small machine. But when MITS expressed interest for his product, he developed this implementation in only 8 weeks, mostly by himself, and it worked perfectly. To date, he says he still remembers all the instructions.

In the eighties, Richard Stallman fought successfully alone against Symbolics, a company which was threatening his idea of free software. He replicated alone what Symbolics developers were producing, and gave access to his source code.

Donald Knuth has been the sole developer of TEX and *MetaFont*, the very sophisticated piece of software used to typeset this document.

Thielen [55] reports the following stories:

> The original Lotus and dBASE were probably the two most successful application programs ever written. Lotus was written by one person in 18 months. Also, the macro capability was added by the developer at the end because he had some extra time — it wasn't even in the informal spec. dBASE was written by one person over a two-year period while he also held a full-time job. Brief (the text editor) was

> written in six months by two developers who had just graduated from college. The Zortech C++ compiler was written by one person in less than one year.

Levy, in "Hackers: Heroes of the Computer Revolution" [33] writes, about the implementation of LISP on PDP-6:

> The crucial sections were written by Greenblatt and another hacker. Two or three people on a project were considered The Right Thing — far fewer than IBM's so-called "human wave" style of throwing dozens of programmers at a problem and winding up with junk.

In the May 2010 issue of Wired, Steven Levy, who authored the 1984 book "Hackers: Heroes of the Computer Revolution" [33], wrote on Mark Zuckerberg, founder of facebook:

> Zuckerberg's adopted style may not come from the golden age of hacking, but his work ethic does. "We didn't start with some grand theory but with a project hacked together in a couple of weeks," Zuckerberg says. "Our whole culture is, we want to build something quickly." Every six to eight weeks, facebook conducts "hackathons," where people have one night to dream up and complete a project. "The idea is that you can build something really good in a night," Zuckerberg says. "And that's part of the personality of facebook now. We have a big belief in moving fast, pushing boundaries, saying that it's OK to break things. It's definitely very core to my personality."
>
> In the ongoing competition for talent, Zuckerberg believes that the company with the best hackers wins. "One good hacker can be as good as 10 or 20 engineers, and we try to embrace that. We want to be the place where the best hackers want to work, because our culture is set up so they can build stuff quickly

> and do crazy stuff and be recognized for standout brilliance."

I have personally seen a single developer, Hakim Erhili, redoing alone the leading product of a software editor in only three months. This was not a small product. His delivery was superior by far, with several orders of magnitude of speed improvement.

I inherited the responsibility of a product with a team of 20 people. I had so many customer complaints on this product that the best solution I found was to replace everybody. I had a single developer, Joseph Canedo, taking over. This was an extraordinary success. All bugs and customer complaints disappeared in a few weeks, and the subsequent commercial success was superb. When I started feeling that leaving a single person on a product was dangerous, as he could break one of his legs or suddenly turn sick, I tried to add another one. That other one was an average one, and the difference between the two was so striking, that the second one, even when he was sitting beside Joseph and watching could not even follow what Joseph was doing. He could not help in any way.

I saw a developer throw in the towel after attempting during an entire year to port a piece of software from X-Window version 10 to version 11. Laurent Chauvin, an excellent developer ported it in a few days.

At some point in time I became in charge of a subsidiary of a technology company. A part of that responsibility entailed data center teams. We had at least one severity 1 every week. Severity 1 was the highest level of incident. Incidents were so frequent and had lasted so long, years, that the teams, about eighty people, were dealing with them as if they were business as usual, except that they spent nights stopping fires. I was lucky enough to find locally an engineer, Terry Moran, who was courted by Google. Given my company's human resources salary guidelines, I could not match what Google offered by a vast margin. Against all odds, as he had never been particularly interested in money, he decided to join the organization I was running. The challenge was probably intriguing. He took care

of the incident issues, and after a couple of months we did not have a single incident. I never saw a single one subsequently. He is now working for Google.

Why do most software professionals have so many similar examples in memory? Discrepancies of productivity of this magnitude are difficult to believe for the ones who have not witnessed them. One of the reasons explaining the skepticism about these stories is that when it is about manufacturing, examples of these discrepancies have never been reported.

Never? The name of Stakhanov is in everybody's minds. Stakhanov was a hero of the Soviet Union. Alexey Stakhanov was a miner, who was said to have outperformed the production norm by a factor of 14. In 1985, The New York Times published an article saying that Stakhanov's feat was only propaganda, and that the record had been "prepared". In 1988, the Soviet newspaper Komsomolskaya Pravda claimed that this performance was obtained by using the help of Stakhanov's colleagues. After all, Stakhanov was simply a marketing tool, conceived to convince the Occident that the Soviet model was capable of stunning performance. 14 times is reputedly not achievable, if the same technique is used[7]. Indeed, every time human movement is involved, like in non specialized pin-making, or even when running 100 meters, variations between individuals are small. On the 31^{st} of May 2008, the world record for running 100 meters was beaten by the Jamaican Usain Bolt, in 9"72. A fair amateur runner completes 100 meters in 14", merely 44% slowlier than Bolt.

Coming back to Stakhanov, given the figures truly acknowledged in software production, 14 times a quota is not a very spectacular record. Software production has its authentic Stakhanovs, even better, and in large numbers.

[7]In the specific case of Stakhanov, as reported by Komsomolskaya Pravda, an increased specialization of labor and task sequencing had been put in place, which might explain in part the performance of the miner.

Figure 1.4: Alexey Stakhanov, front, with his team

1.7 The explanation

This situation is due to the fact that *software production is essentially an intellectual activity*. Intellectual productivity varies greatly from one person to another one. Let us take an example somewhat, even if not fully, related: theorem proving. When examining the performance of different math students, and their ability to build a proof, it can be seen that some are repeatedly two times, three times, ten times, or the infinite times faster than others. Why so? Because theorem proving is an intellectual activity and nothing hampers the speed of the mind. The use of the chalk on the blackboard — I should say a marker now — is a negligible manual intervention. Software production is the same. Software production is of course not only programming, it also involves a great deal of innovation, specification, documenting, testing. But all those are also intellectual activities. Typing on a keyboard is a negligible manual intervention, but most people stop at the surface of things. A computer, a *machine*, is involved, so it must no doubt have something to do with manufacturing.

As early as 1968, Douglas Engelbart, during an incredibly visionary speech now often called the "mother of all demos"[8], while introducing the concepts of personal computer, mouse, hypertext links, version control, project management software, client/server model, groupware, videoconferencing, etc., said:

> The research program that I am going to describe to you is quickly characterizable by saying: if in your office, you as an intellectual worker, were supplied with a computer display backed up by a computer that was alive for you all day, and was instantly responsive[9], how much value could you derive from that?

Engelbart brilliantly envisioned the future with an "intellectual worker" interacting almost constantly with a computer, at a time when the computer was a completely insulated machine people would have access to only occasionally.

Software development is not the only human activity which is sometimes confused with manufacturing. Robert E. Pirsig, in ZAMM ("Zen and the Art of Motorcycle Maintenance: An Inquiry into Values") [40], explains that the very same mistake is made about the maintenance of vehicles:

> Not everyone understands what a completely rational process this is, this maintenance of a motorcycle. They think it's some kind of a "knack" or some kind of "affinity for machines" in operation. They are right, but the knack is almost purely a process of reason, and most of the troubles are caused by what old time radio men called a "short between the earphones," failures to use the head properly. A motorcycle functions entirely in accordance with the

[8] December 9, 1968 at San Francisco's Brooks Hall, now Bill Graham Civil Auditorium.

[9] Engelbart, interestingly enough, stumbles upon this word during his verbal comment of the demonstration and initially pronounces "responsible", instead of "responsive".

> laws of reason, and a study of the art of motorcycle maintenance is really a miniature study of the art of rationality itself.

Or, similarly:

> An untrained observer will see only physical labor and often get the idea that physical labor is mainly what the mechanic does. Actually the physical labor is the smallest and easiest part of what the mechanic does. By far the greatest part of his work is careful observation and precise thinking. That is why mechanics sometimes seem so taciturn and withdrawn when performing tests. They don't like it when you talk to them because they are concentrating on mental images, hierarchies, and not really looking at you or the physical motorcycle at all. They are using the experiment as part of a program to expand their hierarchy of knowledge of the faulty motorcycle and compare it to the correct hierarchy in their mind. They are looking at underlying form.

Motorcycle maintenance, by the way, has a lot to do with software bug tracking. In case there is any doubt left, and you think that the analogy between mechanic work and software development is actually demonstrating that software development should be considered as a lower grade work, you should revise your opinion on mechanic work. Next time you go to the garage and ask for reasonably sophisticated maintenance, check the hourly rate against the one you would pay for a contractor developing software. You'll see how similar they are. And there are several orders of magnitude of complexity between an engine and a piece of software.

Are there any additional reasons explaining why the vast majority of people ignore that stunning productivity discrepancies exist in the field of software production? We shall discuss this specifically, and a lot of reasons can be found. The first that

comes to mind is a simple one, measuring discrepancies of productivity without a formal experimental framework is difficult. As explained earlier, no satisfying definition of production unit exists.

Lutz Prechelt from Karlsruhe University is one of the scarce people who have tried to set a scientific framework [43] for measuring, among other parameters, the relative performance between programming languages, and the productivity of developers using them. Very few studies of the same quality exist, in spite of peremptory claims from a language camp[10] against another one.

This is often due to interpersonal technological affinities discrepancies which bias the results. The affinity of the individuals involved in the test with a given programming language is the dominant factor, not the programming language itself.

Prechelt, in order to resolve this, uses different teams, in parallel, to avoid abnormal data. The result shows, as anticipated, according to Prechelt, that:

> For all program aspects investigated, the performance variability due to different programmers (as described by the bad/good ratios) is on average about as large or even larger than the variability due to different languages.

which lead Prechelt to conclude that:

> Interpersonal variability, that is the capability and behavior differences between programmers using the same language, tends to account for more differences between programs than a change of the programming language.

[10] Donald Knuth prefers the more colorful term *high priests*, which finds its origins far away to the origins of IT at MIT where *priesthood* was a common term used to refer to the limited circle of people who had physical access to the computers. This has been funnily portrayed in the "Hitchhiker's Guide to the Galaxy".

Now trying to measure interpersonal productivity discrepancies is very difficult too. Is anybody able to afford starting the very same project with different teams and check at the end who was more efficient? Microsoft is reported [12] to be able to afford this, not for productivity measurements, but for the sake of the product itself.

Dave Thielen [56] says:

> The single most important contributor to productivity is the quality of the employees. Everything else is secondary to this one criterion.

Similarly, Robert L. Glass writes [20]:

> The most important factor in software work is not the tools and techniques used by the programmers, but rather the quality of the programmers themselves.

While this can be debated, because it puts organization second, it clearly sets the tone. The purpose of this document is to explain that organization and quality of the employees are deeply intertwined, none of them being secondary. It cannot therefore fully agree with Thielen's or Glass' statements, except if we consider the usual organization of work which is indeed the worst possible. But it is a fact that Microsoft tries to recruit the *top 5%*, and makes occasional use of puzzles and tests similar to IQ during interviews, as explained by William Poundstone [42]. Other professional software corporations are reputedly extremely selective, among them Amadeus[11] and SAP[12] to mention two European ones.

Donald Knuth reports a ratio even smaller, of 2% (one out of 50). He said, in September 1996, during an interview by Dave Andrews for Byte Magazine:

[11] http://www.amadeus.com
[12] http://www.sap.com

> We've seen in programming classes that one out of 50 students really grooves on programming. Very few are really into it as top computer scientists. We find that this ratio has held steady for 30 years. [...] I don't look at it as a matter of one person being better than another. Some people are simply going to be able to write much better code [...]

Bjarne Stroustrup, the inventor of C++, Professor and holder of the College of Engineering Chair in Computer Science at Texas A&M University, was kind enough to provide me his view on the same subject:

> I basically agree and I can't really estimate the percentage. Even of the self-selected people who become programmers and/or take programming courses, most will at best become journeymen programmers who laboriously produce poor code that barely do more good than bad. Maybe 10% raise above that.
>
> This is not simply a question of intelligence or effort. Very smart (e.g. good mathematicians) and very hard working people regularly produce lousy (i.e. hard to maintain, hard to use, sometimes buggy, and poorly performing) code.

Thielen indicates that his experience shows that ratios from 1 to 25 can be seen commonly. This ratio can be confirmed by most experienced software professionals.

Robert L. Glass wrote in 2008 in IEEE Software magazine [21]:

> It's well known, if less appreciated, that some people are a whole lot better at building software than others. Scattered through decades of the software engineering literature are studies reporting on "individual differences" ranging from 28:1 (for error identification) to 25:1 (for coding ability) to 11:1 (for timing efficiency) to 6:1 (for sizing efficiency). These

> are differences on the magnitude of 28 times (not 28 percent) better — huge differences. These differences are less appreciated because we in the software field have made little use of this knowledge. As practitioners, we haven't figured out how to identify these 28-times-better people, which might be really important if we need to manage a project that must finish on time and on schedule, or be of ultrahigh quality. As researchers, we haven't sought to acknowledge individual differences when doing experimental research. (For example, how can we be sure, if we're studying a particular methodology's benefits, that it's X% better than some other methodology if the differences between subjects in our study might be 28:1?)
>
> This is one of the most important issues in software engineering, and has been for more than 30 years. I first published an analysis of these individual differences in my by-now badly obsolete book *Software Soliloquies* [19], which was published in 1981. The studies on which I based that analysis dated all the way back to 1968. Of course, I'm not the only one to have written about this matter. I suppose that neither practitioners nor researchers have been able to act on these findings simply because doing something with them falls into the field's too-hard box.

Let us try to do something with them at last. First, given these huge productivity discrepancies, what should recruiters do?

1.8 Beyond salary: full analysis of cost

Many recruiters, especially among *user* companies, neglect the productivity parameter and focus only on the salary. Are they right to do so?

Let us take two developers, the first with a productivity 10 times larger than the second. To exhibit the same cost for the company, given that $C = St$, the productive one should cost 10 times more. Is it the case? Certainly not.

We touch here a fundamental point:

Individual salaries do not vary in the same proportions as productivity.

This paradoxical conclusion, is, ironically, fully in line with Frederick Taylor, who was pushing for reaching a more efficient cost structure by accepting to recruit specific people and to pay them more than ordinary personnel.

As a result, in $C = St$, salary, surprisingly, is the cost parameter which can usually be neglected, as it is today driven by the employment market through vague parameters, and varies least. This result is in complete contradiction with common recruitment practices which focus firstly on salary. Statistically, the people who look "expensive" are actually the cheapest.

Paul Graham wrote in a 2004 essay:

> Economically, this is a fact of the greatest importance, because it means you don't have to pay great hackers anything like what they're worth. A great programmer might be ten or a hundred times as productive as an ordinary one, but he'll consider himself lucky to get paid three times as much. As I'll explain later, this is partly because great hackers don't know how good they are. But it's also because money is not the main thing they want.

It may sound quite cynical, and is probably not, as Graham is a programmer himself.

The difference which has just been highlighted between the software industry or more generally mentofacturing activities and classical industries is in no way related to the idea that mentofacturing is in its infancy. It is related to the nature itself of intellectual activity. A number of people are desperately waiting for the software industry to reach some kind of maturity

and start resembling traditional industries. They will have to wait forever. Software production will never be a conventional industry, it is the first of an *entire* new kind.

Alfred Spector and David Gifford, in "Case Study: A Computer Science Perspective on Bridge Design" [50] said in 1986:

> Bridges rarely fall down. In contrast, computer systems design is one of the least classical of the engineering disciplines, and its products are often poorly understood, unmanageably complex, and unreliable [...]

The temptation to parcellize software production jobs and force in manufacturing practices, will continue to be a recurring phenomenon, as long as its detrimental nature will not be understood. We shall see that software development is intrinsically similar to a craft industry.

As we saw, cost can be drastically brought down by carefully selecting employees, by accepting to pay them sufficiently more than average people. So, when a software corporation is in a competitive situation — and this is most of the time the case —, it has no other choice but to select people who have an excellent intellectual productivity. If the company doesn't, it will not survive ultimately if their competitors understand how important productivity is in the cost equation.

It is understandable that user corporations do not care so much. Software development is a side activity for them. If projects are delivered, they think the cost for software development is affecting them only marginally. While this cannot of course be neglected, it is affecting users far less than it would affect professionals. In reality, user companies are actually deeply affected by the ratio of project failures.

It is interesting to look at what happens to corporations who do not recruit very smart people for software development. If the low tier people are recruited, there is no other solution but turning them into instruments, specialize them, and to introduce inefficient trade-based organization. So, there is no other solution but to create a large number of different trades and to

build a software assembly line. This is, as we shall see, extremely — even exponentially — inefficient, but it is the least dangerous organization when you care about delivering something. On the longer run, specializing people is extremely bad from a human capital management standpoint, as the software industry has a revolutionary nature. Everybody has their own feelings about which technological direction it will likely take. These opinions certainly differ greatly from one person to another. The only general agreement is that technology is bound to change in brutal manners. Specializing people is unfortunately leading them to be inadapted at some future point in time. By the way, the software industry has an insatiable appetite for new people, while rejecting with no pity at all hoards of people on the job market. Many of those people will never find another job again. In this respect, there is a clear and deep moral issue with specializing software staff.

The over sensitivity on salary is also pushing lots of corporations to consider off-shore development. As a number of large professional corporations are quickly developing in lower cost countries, it seems like a good move to many. In the light of productivity discrepancies, from 1 to 25, seeking an average 30% decrease in overall cost (taking into account the costs associated to distance) sounds ridiculous. This is a 30% discount on average programmers, while developers can be found to outperform average programmers easily by a factor 10 in productivity. The efforts Bill Gates makes personally in trying to convince students in excellent American universities to join Microsoft, testify that the growing presence of software giants in lower cost countries must be due to some other reason than simply the cost. Indeed, very smart individuals are, by definition, in utterly limited supply, and to ensure their future expansion, software professionals seek to find in other countries another source of very talented people. And that source exists in developing countries with high levels of population, even if many of the top tier have emigrated to high revenue (and cost!) countries.

1.9 Need for a productivity-friendly organization

Thielen [56] states that the very first contributor to productivity is the quality of the employees. If Thielen meant that superb employees are so bright that they can change the organization if it is not adequate, fair enough. Unfortunately, in many corporations, changing the organization is not easy, and employees fulfill their mission without having the ability or the power to change the way the intellectual production is implemented.

Another remark on Thielen's statement is that however superb the employees are, a bad organization can completely ruin their individual productivity, and hinder all their activities. Earlier in this document, it has been explained that Adam Smith wanted to eliminate the costs related to a person repeatedly changing activity. He explained that a *linear* cost, function of the number of activities, is hidden within non parcellized manufacturing labor. The more activities are at stake, the more costs are associated. In addition, the "focus" of the employees is not sufficient to allow them to have ideas to *automate* their own activity.

Harlan Mills, in his historical paper, "Software Productivity in the Enterprise" published in 1981 [37] wrote:

> There is a 10 to 1 difference in productivity among practicing programmers today — that is, among programmers certified by their industrial positions and pay. That differential is undisputed [...] While this productivity differential among programmers is understandable, there is also a 10 to 1 difference in productivity among software organizations.

Software development has very little to do with manufacturing. First of all the "switching" costs identified by Smith are extremely small when producing software. A developer has many tools at his disposal. These tools, thanks to the multi purpose nature of computers — remember Turing talked about

it as a "universal machine" — are all at hand. On modern operating systems, which are multi tasks, switching from one tool to another one requires typing a couple of keystrokes. Additionally, automation is at root of software development so a developer can modify his own work environment, by himself, and he is used to doing it.

At some point in time, many managers even questioned the value of using a computer to produce software. They believed that continuous access to a computer was not necessary and did not have any influence on productivity. Of course, this seemed like making economic sense, given the price of continuous access to the computer, to share the access with a number of other people, and work only at times on it. But managers believed also that people had to "think" before interacting with a computer, and work on a piece of paper, while others would take their turn to access the live computer. That was forgetting the fact that a computer is so universal, that it can even provide a far better solution to edit a design or a program than a piece of paper, a pencil and an eraser. The fluidity, and the help you get by interacting with a computer is far greater than with any concrete tools. This even fascinated the early hackers. Steven Levy, in [33] says:

> The challenge of programming appealed to Gosper. Especially on the PDP-1, which after the torture of IBM batch-processing could work on you like intoxicating elixir. Or having sex for the first time. Years later, Gosper still spoke with excitement of "the rush of having this live keyboard under you and having this machine respond in milliseconds to what you were doing...".

Adam Smith "tools switching" linear cost is therefore negligible in the software industry. But, in that industry, the difference between software production and manufacturing brings two additional costs, which are also hidden costs, not obvious to all immediately.

1.9.1 Linear cost related to information exchange

When confronted to a functionally rich problem to solve, software construction is very often organized around an assembly line. It is obvious to all that an assembly line is not required if the problem fits in a regular programmer's head. In that case, everything is handled by the same person.

Within an assembly line, given the complexity of even a moderate size project, a lot of information transfer needs to take place. Producing software is an innovative process, so knowledge is constantly evolving. This is very different from a manufacturing assembly line, in which the material passed from one trade to another one is stereotypical. Information transfer is therefore a completely new factor, brought by the nature itself of software production.

Information transfer may occur between the architect trade and the database trade, or between the database trade and the trade responsible for server-side code, or between the server-side trade and the user interface trade.

These are only examples, of course, but they illustrate how massive information transfer can be. It is to be noted that that information transfer is mandatory. A general and unique information distribution to the various trades in a single shot is not possible, as the interaction between what the trades produce is very low level, beyond what specification can reasonably reach.

Information transmission costs and issues exist also in classical engineering activities, although they are often marginal. Alfred Spector and David Gifford in "Case Study: A Computer Science Perspective on Bridge Design" [50], where they compare civil engineering of bridges and software development quote Gerard D. Fox:

> There aren't an enormous number of people involved in the design; when there are too many they get in each other's way, and it becomes more difficult to keep everyone up-to-date with changes.

Fox talks about bridge engineering, his specialty. Informa-

tion exchange in civil engineering disciplines is not a dominant factor, as design teams are small. But it is a dominant activity when developing software. We have to remember that design is the only activity involved in software development. Everything else is automated. It is like putting a magnifying glass on the small design team Fox talks about.

Frederick P. Brooks, in "No Silver Bullet: Essence and Accidents of Software Engineering" [8], stated that:

> There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity.

As we shall see in section 1.12, statistics show that software projects success ratio falls roughly exponentially with their size. So there is plenty of room for improvement. Brooks makes a risky bet, because we are in such a disastrous situation already.

At the same time, Brooks insists on a point which is surprisingly not highlighted in Spector's paper, complexity:

> Software entities are more complex for their size than perhaps any other human construct because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into a subroutine — open or closed. In this respect, software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.

he adds:

> The complexity of software is an essential property, not an accidental one.

and

> Many of the classic problems of developing software products derive from this essential complexity and its

> nonlinear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, schedule delays.

While it may be debated whether software complexity grows non linearly with its size, information transfer between team members is clearly identified as one of the key problems to resolve.

The cost for information transfer is a linear function of the number of trades. Therefore, the division of labor in software production introduces a linear cost related to information transfer. That linear cost was absent in manufacturing.

That cost is one of the reasons explaining why, as Brooks said, assigning more programmers to a project running behind schedule, may delay it even more. Adding staff is a very costly operation, in terms of knowledge transfer, and may further hamper the organization.

We need to remember that the linear cost Adam Smith identified when workmen switched from one machine to another one, was, by itself, decisive for supporting *maximal* division of labor, and deeply changing manufacturing during more than one century.

In the software industry, as we just saw, a linear cost exists also, but this time, it is *defavorable* to the division of labor. This alone, should push to implement a *minimal* division of labor compatible with a proper production. This is true for division of labor in the same sense as Adam Smith, which translates into the creation of different trades. The smallest amount of trades reasonable should be used. It is also true from a project standpoint, because, even if it is obvious that many projects exceed the capability of a few people, and project needs to be split, the least division is implemented, the more cost effective the organization.

There is more to come: behind the linear cost we just identified, lies a more pernicious one, which is exponential.

1.9.2 Exponential cost related to information exponential entropy over the assembly line

The hidden exponential cost that lies right behind the linear cost we just exposed is also related to information transfer. It is therefore absent from manufacturing, as no information transfer occurs between manufacturing trades.

To be fully accurate, some similar exponential cost theoretically exists even in manufacturing, although usually negligible. In a manufacturing assembly line, each trade may introduce errors, mistakes. Over the chain, theoretically, these mistakes could cumulate, producing ultimately a result which combines the various mistakes in an exponential manner. In practice, such an exponential combination of mistakes is easily spotted when it occurs, as the assembly line manipulates stereotypical objects.

Software production is a very different matter. The bulk of information is enormous. We already said it is not stereotypical, but the fruit of a constant innovation. Misunderstandings and mistakes introduce a great deal of noise along the assembly line. Just because the solution is too bulky to fit into a single person's head, there cannot be a super controller checking that all information transmission is correct at each step. The "noise" which is introduced brings a gradual entropy that cumulates until the last step. The delivery of the assembly line can be heavily affected by that. The more mediocre the staff is, the more entropy is created. As assembly lines are usually made of the low tier developers, entropy is an acute issue.

To be fully accurate, if we want to estimate the cost we are talking about, let us make the following assumptions in our model, to simplify calculation:

- The division of labor is applied over `n` different persons.
- Work is equally divided among the staff. Each person is therefore doing a share of the work equal to $1/n$.
- Each person is introducing an entropy ratio of α.

The first person introduces α amount of error to his $1/n$ part, which means a volume of $\frac{1-(1-\alpha)}{n}$ errors. The second one, in sequence, adds a combination of his errors and the ones brought by the former one: $\frac{1-(1-\alpha)^2}{n}$, etc. up to the last one, who adds $\frac{1-(1-\alpha)^n}{n}$ errors.

The total amount of errors, in other words, the *entropy* is:

$$\sum_{i=1}^{n} \frac{1-(1-\alpha)^i}{n}$$

Which is equal to:

$$\boxed{1 + \frac{(1-\alpha)^{n+1} - (1-\alpha)}{n\alpha}}$$

We can safely qualify this quantity as approaching 1 exponentially when the number of trades grow, even if theoretically it is not purely an exponential (`n` can also be found on the denominator). Indeed, the number of trades appears at the power of an exponential, increasing — theoretically — pretty quickly (even with the number of trades on the denominator), with an asymptotic value of 1 (meaning 100% of errors). Figure 1.5 shows the result for a few different values of α. In practice, reasonable values for the number of trades scarcely excess 10. The part of the curve we are interested in looks more like an exponential when the values of α are high. But even when the curve is quasi-linear, its slope is very worrying, and entropy reaches fast unacceptable levels. A teaching derived from the interpretation of this curve is that to avoid an unreasonable amount of entropy, staff must have very good aptitudes at communicating together, both at explaining and understanding. It contradicts with the type of personnel you get when division of labor is put in place.

A lot of project methodologies, like Kent Beck's now long defunct *Extreme Programming* [4], or more generally the "Agile" family, among which Scrum for instance, try to fix the entropy issue, and propose to involve the customer throughout the construction process. Unfortunately, in general, users are

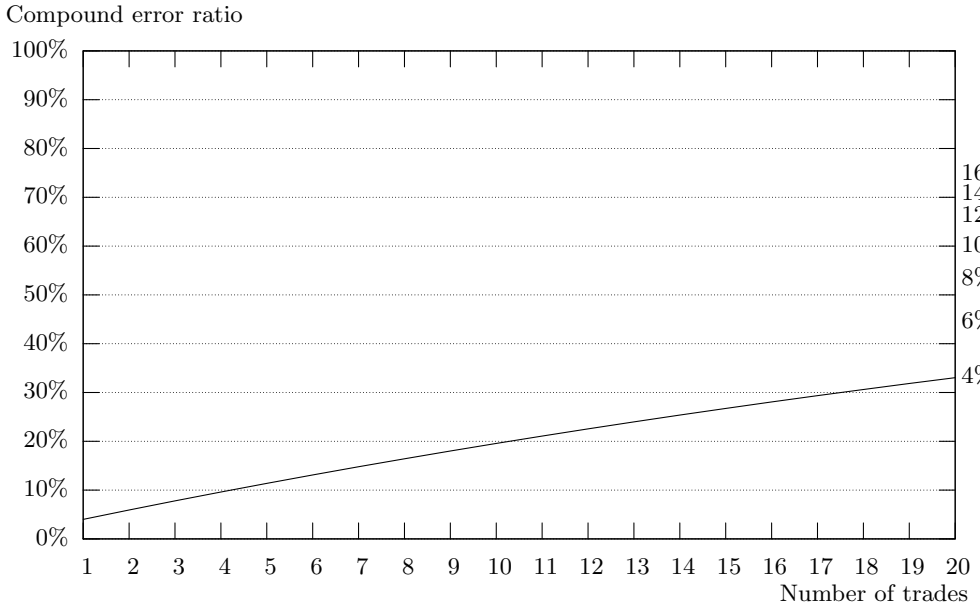

Figure 1.5: Compound error ratio as a function of the number of trades. The lowest curve corresponds to an individual trade error ratio of 4%, and each of the subsequent upper curves correspond to 2% increments.

not fluent with technology, and they are unable to validate each of the steps. It has led "agile" methodologies to introduce the idea of iterations, from mock-up to various levels of prototypes, to repeatedly show to the customer the solution being shaped. This way, the non technical customer can actually *judge* if the solution goes towards his expectations. While it ensures convergence, going repeatedly through the assembly chain remains very expensive, and progressing by making mistakes and backtracking frequently leads to high costs. There is no surprise that these methodologies are only applicable to smaller projects, where entropy is not too significant, and where success is more important than cost. In a small team, most players can easily share infor-

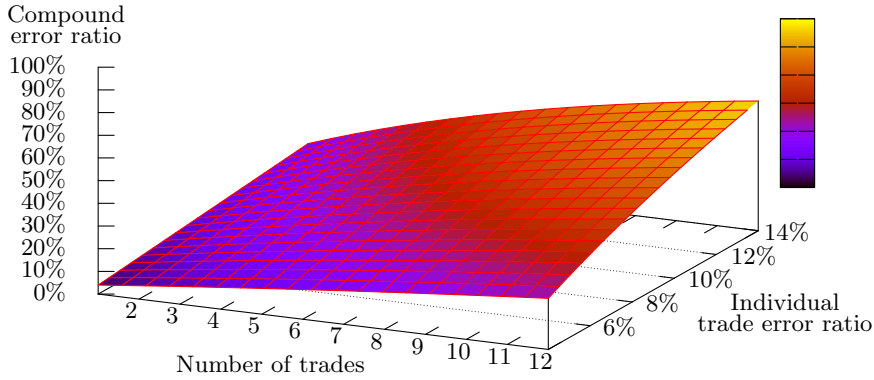

Figure 1.6: Compound error ratio as a function of the number of trades. Surface view.

mation. Agile methodologies are also applied to projects where requirements are unclear, and require iterations, by definition. As a consequence, the core issue coming from interpersonal dependencies is not solved, but simply hidden.

Involving an *architect* group along the project, as proposed by Brooks, is somewhat alleviating the burden of entropy, but, given the level of detail these architects would need to reach in order to remove it completely, the size of such a group would quickly raise worries about the redundancy between programmers and architects. It would also create the same issues of sharing knowledge between the architects, moving the problem instead of fixing it.

1.9.3 Fixing linear and exponential costs

From the two former paragraphs, it is obvious that, to reduce software production costs, it is important to reduce *dependencies* between personnel. These dependencies did not exist in manufacturing organizations, and constitute a new factor to take into account. Conversely, tools switching costs do not exist when producing software, thanks to Alan Turing's universal machine, which is the base of all modern computers. The rules underlying software production are, as a consequence, totally different from the ones which prevail in manufacturing.

In case software production is organized using the division of labor, fixing the exponential entropy means fixing *a posteriori* the delivery of the assembly line. This cost is linear with respect to the amount of damage, and that damage is exponential with respect to the number of trades.

An organization that would avoid this entropy would exponentially improve developers' productivity. Such a process would contradict Brooks [7], who said that no silver bullet existed for enhancing developers productivity, and that there was no single strategy, technique or trick that would exponentially raise the productivity of programmers.

That organization exists, it is the one implementing *minimal dependencies* between personnel. Cutting down projects translates into splitting along least dependencies cleavages. The division of labor almost always splits along maximal dependencies cleavages.

In practice, except in extreme cases, minimal dependencies imply *minimal division of labor*, or in other words, minimal specialization. Surprisingly, the conclusions that Adam Smith reaches about *maximal* division of labor for manufacturing activities, are completely opposite to the sensible recommendation for software organizations:

An organization producing software should reach minimal *division of labor in order to be efficient.*

As a corollary, software production should not be implemented using an assembly line.

Least parcellization does not equate to no parcellization at all. A number of reasons push for *controlled* parcellization:

- The lack of people combining several skills may lead to dividing labor in order to allow recruitment.
- The need for *independent* decision-making may lead to split activities between different trades.
- The necessity to build homogeneous solutions may lead to have separate groups to check the various aspects of homogeneity.

Scarcity of some skills combinations

An illustration of the scarcity of some skills combination is the following one: it is rather difficult to find people who combine excellent mastering of the software development *art* — in the sense of Professor Knuth [29] — with classical artistic talents. Would an organization be able to find a source of talented people combining the two, it would not be necessary to split activities requiring software development and artistic work. Practically, though, such a source of talented people is difficult to find, and a real need for separating *mostly artistic* trades arises. As an example, Web Designers may collaborate with pure software developers. Similarly, not all developers are able to understand ergonomical constraints, and specialists of ergonomy may be necessary. It needs to be noted that the lack of combined skills can very often be resolved by appropriate training. Even Taylor insisted on the importance of training on productivity. Smart people can absorb very large amounts of information and have a natural ability to manipulate very diverse abstract matters. Ergonomy can be taught, there is good litterature on the subject [10], and in case the developers can add this skill to their capabilities, it is far more efficient to avoid creating specialized ergonomy groups, and add another skill to the developers instead. But the closest you get from graphical arts, the toughest it becomes to train every person. It can be argued that artistic

skills cannot be taught to everybody and technique plays only a minimal although necessary role.

Independent decision-making

A separate group is required when a separate decision is to be made, when a conflict of interest arises. It is a very bad idea to have people being the judge in their own case. Different organizations are therefore mandatory when concerns are different. This establishes organizational boundaries between different groups, with independent standpoints, which must not be coupled. As an example, having marketing defining the prices, and sales making deals is an example of concern separation. If sales were defining prices, they could be tempted to lower them to make their life easier, and their goals more easily reachable.

As an other illustration of this, let us take the example of software corporations which go beyond the classical model of software editors. Those corporations endorse responsibility beyond the point where they deliver software to their customers. They both maintain software solutions and operate them, with contractually guaranteed service level agreements with their customers. Among those corporations, you can find *Global Distribution Systems*, such as Amadeus for instance. GDSs offer facilities for travel reservations, on behalf of travel providers like airlines, hotel chains, car rental companies. The operation of the software is a critical piece of the service offered by GDSs, and Amadeus has the largest non military datacenter in Europe, all industries being considered.

The case of this type of corporations who deliver a complete service is interesting because they are the first significant players of a new era within the Information Age. They belong to industry *vertical* specific software solution providers. The new era they belong to could be qualified as the *Vertical Age*, which followed the *Commodities Age*. The Commodities Age saw the birth of software giants such as Microsoft and Oracle. The *Vertical Age* corporations, instead of exonerating themselves from any responsibility towards their customers, engage their respon-

sibility in terms of amount of bugs or production-related availability of their software. In some occasions, these commitments lead them to pay enormous eight figures indemnities to their customers when an outage occurs. The commitment to pay indemnities pushes for a far higher amount of testing than the one usually done for off the shelf software.

This situation is very different from the one of Microsoft for instance, which does not guarantee quality with its own assets, and risks only its reputation when too many bugs arise. As explained by Thielen [56], Microsoft, like many other software editors, spends the exact amount of testing so that their software is accepted by the market. Microsoft is therefore spending the exact amount of resources, no more, no less so that there is no risk of being displaced by competition. As a consequence, Microsoft is sometimes even building business cases with its customers to fix problems. Only when the business cases are positive enough, will the bugs be fixed. This is often mistaken with a suspected inability of Microsoft to deliver high quality software. As a matter of fact it only reflects the strength of Microsoft position. We are indeed very far from Knuth's legendary exponentially growing reward[13] for finding TEX bugs, and the obsession to deliver mathematically proven software, exempt of any bug.

The corporations which spend a large amount of their resources in testing, have put in place cascading test systems, operated independently from each other. The software product percolates through several successive systems, and when a problem arises, the product is sent back to the engineering staff. When the problem is fixed, percolation starts back at the be-

[13] Donald Knuth wrote the following about TEX:

> I believe that the final bug in TEX was discovered and removed on November 27, 1985. But if, somehow, an error still lurks in the code, I shall gladly pay a finder's fee of $20.48 to the first person who discovers it. (This is twice the previous amount, and I plan to double it again in a year; you see, I really am confident!).

Note that Don Knuth is a real computer scientist, $20.48 is 2 cents risen to the power of 11.

ginning. Some test systems require human intervention, and a human decision to evaluate risk. It is very important, in order to avoid parasitic effects, that these human beings make their decision independently. For those reasons, in many corporations, a number of trades are dedicated to testing, are independent from the development workforce, and work independently from each other. This leads to a necessary parcellization of work.

The need for homogeneity check

When complex homogeneity requirements are to be met, a separate role may be created to endorse that responsibility. An example is the need to have consistent graphical user interface behavior. In that case, when this commonality cannot be enforced by a technical framework, a separate human responsibility is needed. Another famous illustration can be found in Brooks [7]. In "The Mythical Man-Month" [7], Brooks explains that correctly architected solutions, which make sense as a whole, require a separate architect group. While this is still followed by many professional software corporations, including Microsoft, it must be taken with great care. First, it applies only to a subset of the software industry, which produces solutions that differ meaningfully from each other, and require a different architecture most of the time. This is true in all technical products, such as the ones IBM or Microsoft are developing. This does not hold for the solutions developed for vertical markets for instance. Software solutions share the same blueprint of technical infrastructure, and reuse it over time, until the next technological evolution, when architectures are revisited. Also, it does not hold for corporations which directly map technical responsibility and managerial responsibility. In this case, the management line is the group of architects.

1.10 From Programmers to Developers

If *minimal* division of labor needs to be used as one of the organization principles for software production, how is it possible

to cope with large scale systems? Many projects exceed the capabilities of a single person, even a very productive one. Large projects require that work is split among a number of people. What can be the rules to cut the project down to manageable sizes a single person can cope with?

The solution is to divide the project in an orthogonal manner with respect to the classical division of labor and trades. Boundaries should not be technological ones, but functional ones. From horizontal divisions, the divisions become vertical. As a result, the number of trades should be minimal, and a given person, in one trade (for instance specification development or code development), should cover a portion of a product, instead of a portion of technology. This portion of a product should span all the technological aspects, for instance: database, server-side code development, graphical user interface, network, and documentation. If several different programming languages are to be used, the very same person will be exposed to all of them. We can refer to this project split as the *Functional Division* method, by comparison to the *division of labor* method, which uses technological boundaries as cleavage points.

This would please Ralph Waldo Emerson, who criticized the parcellized society inherited from Adam Smith. He praised the "Man Thinking" in his famous speech "The American Scholar" [16] delivered before the Phi Beta Kappa Society, at Harvard in August 31^{st}, 1837:

> The old fable covers a doctrine ever new and sublime; that there is One Man, — present to all particular men only partially, or through one faculty; and that you must take the whole society to find the whole man. Man is not a farmer, or a professor, or an engineer, but he is all. Man is priest, and scholar, and statesman, and producer, and soldier. In the divided or social state, these functions are parcelled out to individuals, each of whom aims to do his stint of the joint work, whilst each other performs his. The fable implies, that the individual, to possess himself, must

> sometimes return from his own labor to embrace all the other laborers. But unfortunately, this original unit, this fountain of power, has been so distributed to multitudes, has been so minutely subdivided and peddled out, that it is spilled into drops, and cannot be gathered. The state of society is one in which the members have suffered amputation from the trunk, and strut about so many walking monsters, — a good finger, a neck, a stomach, an elbow, but never a man.
>
> Man is thus metamorphosed into a thing, into many things. The planter, who is Man sent out into the field to gather food, is seldom cheered by any idea of the true dignity of his ministry. He sees his bushel and his cart, and nothing beyond, and sinks into the farmer, instead of Man on the farm. The tradesman scarcely ever gives an ideal worth to his work, but is ridden by the routine of his craft, and the soul is subject to dollars. The priest becomes a form; the attorney, a statute-book; the mechanic, a machine; the sailor, a rope of a ship.
>
> In this distribution of functions, the scholar is the delegated intellect. In the right state, he is, Man Thinking. In the degenerate state, when the victim of society, he tends to become a mere thinker, or, still worse, the parrot of other men's thinking.

When a person, a *Man Thinking*, is entrusted with an entire functional subset, his intellectual productivity can deploy fully, without being hampered by the division of labor. The most productive endorse larger projects, which do not need to be subdivided further, and can truly express their productivity, allowing the organization to have a maximal efficiency.

In an assembly chain, as identified by Taylor, chronometers must be used to fine tune each trade so that a given person does not wait or is not waited for too often. Chronometers allow also to monitor productivity. When minimal specialization is applied

in a "mentofacturing"[14] activity, given the fact that subprojects are divided alongside minimal inter personal dependencies, the potential loss of time is reduced by properly allocating tasks according to the extent of the task and individual productivity. When somebody finishes early, he can take a subproject (again selected with minimal dependencies) from one of his colleagues' areas, usually a connex one. It leads to a situation where no loss of time is induced.

When considering the code development activity, we are, as a consequence, not talking anymore about a *programmer*. "Programmer" was used at the time of labor specialization along with other software-related job names, so it is itself a trade designation, which is heavily carrying a division of labor ideology. The appropriate term should be *Developer*, as it implies a wider spectrum of expertise and an extensive responsibility. The word "programmer" should be banned. Some people try to fight its negative connotation by insisting on using it, we must wish them success.

Clearly, only very smart people can become developers. People who were used to being specialized programmers (this is pleonastic), and who have the ability to become a developer are scarce. As minimal parcellization of labor is meant for very productive people, and only productive people can sustain minimal parcellization, very selective recruitment which is meant to detect productive people, and organizations implementing minimal division of labor, are completely interdependent. The very same idea applies to autonomy. Minimal parcellization pushes for allowing people a great deal of autonomy, and only smart people can be trusted, and offered this kind of autonomy. The two are intertwined.

The principle of cutting problems along functionalities applies not only to a team, but, even more, to entire organizations. Departments dedicated to a given technology slice should be avoided, in order to reach a better global efficiency. Again, as indicated before, having dedicated teams for consistency coordi-

[14] "mentofacturing" is a neologism, which means *made with the mind*, by opposition to "manufacturing", which comes from *made with the hands*.

nation is an option to be considered, if technological frameworks are not sufficient to ensure the coherence of the solutions.

1.11 Positive consequences of minimal division of labor

In addition to maximizing productivity, minimal division of labor has a number of advantages. Minimal parcellization allows jobs with a broader spectrum of expertise to be offered. Only this makes it possible to keep the smartest, by offering them a difficult challenge, and the possibility to enrich their skills. Of course, the more skills people have, the more opportunities they have to leave the company and easily find another job. The challenge for the company, as a consequence, is to keep their personal development at a higher pace than what they would get in another corporation. Quickly, people have all the possibilities to leave the company, and the company must ensure that the staff do not feel like doing it.

Being exposed to a variety of technologies is also an excellent way to be resilient over time. We said that software is a continuous revolution, with brutal discontinuities. Those technological discontinuities arise regularly, we can, just like a future earthquake or volcano eruption, even *count* on them to happen! Specialized people do not resist well to those discontinuities, and their adaptation to something new is both difficult and painful, when it is at all possible. There is, by the way, a moral issue with the promise that many recruiters make about being able to propose a long term role to people with small skills. Those will end up having to switch from IT to the least likely professions, such as nurse, gardener, or taxi driver. I have personally in mind the case of the CIO of a large corporation who was fired, never found back any job in the IT industry, and ended up as a taxi driver.

Unfortunately, people tend to prefer specialization. It brings a comfortable situation, if not often a prominent position. Linus Torvalds, the inventor of Linux, thanks to his experience man-

aging a big team of voluntary people, says the following about "Kernel Management":

> One thing to look out for is to realize that greatness in one area does not necessarily translate to other areas. So you might prod people in specific directions, but let's face it, they might be good at what they do, and suck at everything else. The good news is that people tend to naturally gravitate back to what they are good at, so it's not like you are doing something irreversible when you *do* prod them in some direction, just don't push too hard.

In order to avoid too much specialization, a manager must push *hard enough*.

Avoiding specialization, also allows to avoid technological fanatism, which is a very frequent syndrom among developer groups. But we'll talk about this in the next chapter.

As we saw earlier, minimal division of labor and the intertwined recruitment principles allowing it, also create a higher degree of delegation. Smart people are empowered and keep a very high motivation. This contributes greatly to staff retention.

Lastly, when a corporation manages diverse project sizes, with consistent use of technology across projects (this is common sense, given the economies of scale it brings), developers who are experienced in minimal dependencies organizations have no difficulty moving from small size projects to larger ones, as they are not specialized in a subset of the tools required.

The division of labor is never uniformally applied. If it was, smaller projects would become very costly, as they would involve specialized developers and information transfer costs would constitute the majority of the costs. As a consequence, usually, small projects are managed with non specialized developers, while specialized developers deal with larger projects. Two populations coexist. The ones belonging to the first population can join the second group, while the second type cannot take care of small projects because of a lack of skills.

Organizations which uniformally divide projects with minimal dependencies favor mobility.

1.12 Organization and projects success ratio

It is interesting to note that classical division of labor applied to software development organizations has a meaningful impact on projects success ratio. Also, this impact varies depending on the size of the project. It is obvious that success probability should slightly decrease for projects which are larger, for a number of obvious reasons. But as a matter of fact, so many large projects fail that it exceeds intuition by far. This has always been a matter of intense debate. The most famous paper on the subject has been written by Alfred Spector and David Gifford [50], and is entitled "Case Study: A Computer Science Perspective on Bridge Design". It compares civil engineering of bridges to software projects development:

> Bridges rarely fall down. In contrast, computer systems design is one of the least classical of the engineering disciplines, and its products are often poorly understood, unmanageably complex, and unreliable.

The division of labor is an organization we are all used to. Actually it is the only one that comes naturally to mind. It is not a natural one, and was pretty "out of the box" when Adam Smith proposed it during the middle of the eighteenth century. Unfortunately, when applied to software projects, it asphyxiates them. Figure 1.7 highlights this. It is extracted from the CHAOS research, a yearly report by The Standish Group. This report has always been the object of an intense and heated debate, because a lot of people question its authenticity. It is indeed quite shocking. If you believe it, you do not start a significant software project. The Standish Group is rather silent on its methodology and no peer reviews exist. Nevertheless, it is worth a look with a healthy doubt. It is true that experience

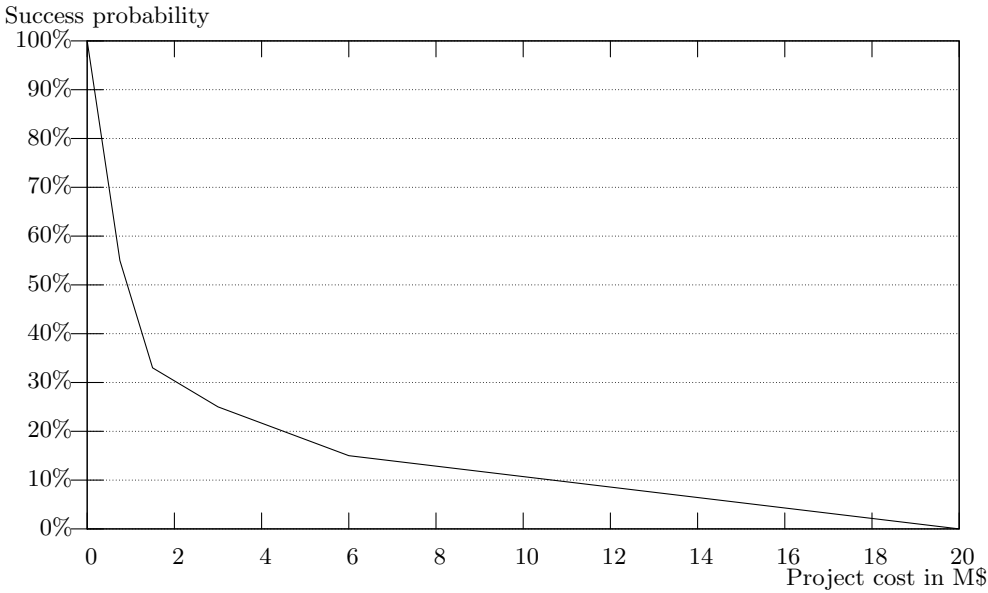

Figure 1.7: Project success probability as a function of project size. Source of data: The Standish Group International, Inc. CHAOS 1999 research.

confirms that the larger the projects, the bigger the risks of failure, and in a quite steep manner.

Even in the small projects area, which is the leftmost part of the curve, the chances for success are rapidly very low, simply because the project collapses under practical linear and exponential efforts (see sections 1.9.1 and 1.9.2).

For zero size projects, obviously, all project division techniques have a sucess probability of 100%. The curves coincide.

Starting from the leftmost part, and going to the right, the curves at first are very similar. Single person projects behave exactly the same. Later on, for multi persons projects of a small size, the division of labor starts dropping immediately. For small projects, people are in the same office, and dependencies do not

translate into costs which are too high. Soon, for bigger projects, people are scattered on an entire floor, then entire building, and in extreme cases, throughout different geographies, and potentially the entire world. The success ratio of the division of labor is deeply affected by these unavoidable phenomenons. Functional division is also impacted by the risks undertaken in larger projects, but in a milder manner.

Functional division evolves smoothly, for the simple reason that it promotes splitting projects into subprojects. The probability of success is the combination of smaller size project probabilities, affected only in a light manner by some dependencies. Each person takes care of a variety of technologies, like in craft industries (see page 44). The organization morphs smoothly from small to very large projects. So does the probability chart. In addition, the type of personnel employed in the two organizations is drastically different, allowing for a higher success ratio in the case of functional division. The division of labor introduces a discontinuity in the organization. When projects are small, they are handled by a single person doing all, and when larger projects are to be divided, each person is attached to a tool, and the split over different people is suddenly done in a different way.

1.13 In search of the exceptional man

In a nutshell, given what was said in the former part of the document, it is necessary to look for exceptional people, what Taylor himself designated as the "Exceptional Man". The title of this section is intentionally reusing the same expression. Nowadays, to be politically correct, and to avoid sounding sexist, I should have used "Exceptional person"...

Why isn't the need to search for exceptional persons acknowledged more widely? There are very old barriers against that, including from Adam Smith himself.

Thirty five years before the Wealth of Nations was first published, in 1741, the danish satirical author Ludvig de Holberg published a story, in latin, called "Nicolai Klimii Iter Subterraneum"[26],

which stands for "The subterranean journey of Nicholas Klim". In this book, Holberg's hero, Nicholas Klim, falls into a subterranean world, where he meets the utopian nation of Potua, formed by apathic creatures half human half tree. As a consequence, their culture makes the apology of slowness, and they condemn rapid pace of thoughts. Holger wanted to criticize some of his contemporaries who preferred immobility to velocity.

As for Adam Smith, in order to help justify his thesis, and probably for ideological or moral reasons, he insisted on the fact that natural talents are of small importance. In the Wealth of Nations, he tried repeatedly to minimize natural individual discrepancies. Let us go back to the text:

> The difference of natural talents in different men, is, in reality, much less than we are aware of; and the very different genius which appears to distinguish men of different professions, when grown up to maturity, is not upon many occasions so much the cause, as the effect of the division of labor. The difference between the most dissimilar characters, between a philosopher and a common street porter, for example, seems to arise not so much from nature, as from habit, custom, and education.

Examples are numerous, here is another one (he did not seem to like philosophers very much... Actually, he taught philosophy, so it must rather be interpreted as modesty):

> By nature a philosopher is not in genius and disposition half so different from a street porter, as a mastiff is from a grey-hound, or a grey-hound from a spaniel, or this last from a shepherd's dog.

He sporadically acknowledges that superior talents exist, but he is still very uneasy with that, as shown here (now he likes philosophers):

> The over-weening conceit which the greater part of men have of their own abilities, is an ancient evil

> remarked by the philosophers and moralists of all ages.

After all, Adam Smith, before publishing the Wealth of Nations, was professor of moral philosophy at Glasgow University.

Frederick Taylor, in contrast, states initially that instead of desperately searching for the exceptional man, corporations should rather revise a few of their principles for managing their staff. However, he clearly indicates that exceptional men are even more necessary with his proposals. He even openly talks about carefully *selecting* employees.

Knuth, who reported a small 2% of his students being able to groove at programming, feels like balancing this statement with the following:

> I don't look at it as a matter of one person being better than another. Some people are simply going to be able to write much better code, but their code isn't necessarily going to be the better system for someone who doesn't think like the programmer.

In modern times, the perception with respect to smart people has evolved, especially among software professionals. But a lot of contrast exists between professionals and the rest of the world. Thielen, an ex-Microsoft employee, highlights that when he publicly explains that the quality of the employee drastically changes productivity, this is often met with *open hostility*. He gives two explanations: the role of the manager needs to be redefined and the manager is no longer in control, while the employees are.

The range of reasons which restrain people from recruiting smart employees is even more complex:

1. Traditional organizations need to be revisited, both to adapt the role of management and to ensure that processes are adapted to not hampering highly productive people.

2. Managers are afraid to loose their job if they recruit people smarter than themselves.

3. There is a wrong perception that recruiting very smart people will create career management issues. Retention of staff appears difficult in the mid or long term.

4. "Elite" in software is often synonymous to hacker.

5. As indicated by Randall Stross [51], there is a rampant anti-intellectualism that does not shed a positive light on particularly smart people.

6. Rewarding people on the base of their more or less natural intellectual abilities, raises a moral issue.

1.13.1 Reforming organizations

Organization is the least easy to change parameter in a company. People get used to their role, and try to reproduce it constantly. When the organization changes, especially when it deeply reforms people's roles, a very high level of inertia or even resistance is to be expected.

As we saw earlier, minimal parcellization pushes for delegation. This pushes to revisit the role of management. Instead of naively applying the army model of management, with people *executing* their task, people endorsing managerial responsibility have to learn how to deal with a high level of autonomy from their staff. The ethymology of the word "execution" is, by the way, interesting, as it comes from the latin word *exsequi*, which means "follow". Productive people are not followers, they are all leaders of change and innovation. This is why they usually clash with traditional management style.

Minimal division of labor is also extremely demanding for managers, as they should not evolve into administrative personnel, and must be capable of "riding the bull", while keeping the right level of control and decision-making, necessary for the success of the projects.

1.13.2 Job protection

Linus Torvalds, who is the head of the group maintaining Linux, his own initial invention, says in a document on "Kernel Management": "Some people react badly to smart people. Others take advantage of them". This is an excellent summary. However, he is slightly wrong, and overestimates most of the managers. Actually, *most* people react badly to smart people. Also, "taking advantage" is not a very positive expression, but in any case, it is a very striking statement. Linus certainly has a very accurate feeling about managing smart people, simply because nothing can be more difficult to manage than a team of independent smart people contributing voluntarily to free software.

Productive people are found among the smartest ones. What is feared most by a manager recruiting staff? Instead of — legitimately — being afraid to fail their project, a manager usually fears being endangered by people smart enough to challenge them, or potentially have greater ideas than theirs. This is why job protection is one of the strongest factors which fight against minimal parcellization. This factor explains in part the reason why classical manufacturing organizations are desperately enforced by many managers in software organizations, because they would hate to recognize interpersonal productivity discrepancies. The best people simply compensate for the worst ones the way they can, given the organizational constraints they are forced to obey.

One of the other consequences of job protection is the "rotting branch", or "bozo explosion" syndrom. Not so smart managers recruit even less smart subordinates, who in turn hire even worse staff. The hierachical line is like a branch, which rots down to the leaf. Former Apple evangelist, Guy Kawasaki, says about his former boss, Steve Jobs:

> Actually, Steve believed that A players hire A players — that is people who are as good as they are. I refined this slightly — my theory is that A players hire people even better than themselves. It's clear, though, that B players hire C players so they can feel

> superior to them, and C players hire D players. If you start hiring B players, expect what Steve called "the bozo explosion" to happen in your organization.

Controlling that the branch does not rot, and helping managers to remain secure and feel safe while recruiting very smart individuals is a very important aspect of corporations who deliver intellectual services. Working with brilliant people is a reward of all moments, as Bill Gates underlines it frequently. This is one of the reasons why the richest man on earth remains very motivated.

Linus Torvalds, the original author of Linux, has a interesting piece of advice about managing bright people:

> So when you find somebody smarter than you are, just coast along. Your management responsibilities largely become ones of saying
>
> "Sounds like a good idea — go wild", or "That sounds good, but what about xxx?". The second version in particular is a great way either to learn something new about "xxx" or seem *extra* managerial by pointing out something the smarter person hadn't thought about. In either case, you win.

And about fighting the feeling of uneasiness managers get about their *legitimacy*, that even himself obviously felt:

> First off, while you may or may not get screaming teenage girls (or boys, let's not be judgmental or sexist here) knocking on your dressing room door, you *will* get an immense feeling of personal accomplishment for being "in charge". Never mind the fact that you're really leading by trying to keep up with everybody else and running after them as fast as you can. Everybody will still think you're the person in charge.
>
> It's a great job if you can hack it.

1.13.3 Retention and career management issues

When recruiting brilliant people, you can expect to face a number of expectations, either in terms of reward, or in terms of career deployment. These expectations, combined with an unusual density of talents, seems to produce a difficult situation where you cannot reward everybody, and not everybody can access managerial positions.

Brilliant people create brilliant solutions, and if the remainder of the organization is able to take benefit from it, by aiming valuable markets and by succeeding in selling at a good price, reward to everybody is not an issue.

Now, about career management, the answer is more subtle. Let us take the opposite situation to set everything clear. If low profile people are recruited, they will fail to deliver a number of projects. Some of the projects delivered will look so bad, that customers will complain. Your competition will certainly recruit smart people, and you will fall inexorably into slow or quick business shrinkage.

If very talented people are recruited, with an excellent productivity, all projects will be delivered and they will exceed customers' needs. More customers will be attracted, because thanks to your average staff productivity, prices will be kept low. Your business will grow exponentially, and career management will not be an issue anymore. Actually, it can be, as I experience it regularly, the other way around, that is that the business is growing so fast, that the company cannot find internally enough people sufficiently experienced to fill the open positions.

In case not everything goes well, in spite of having recruited superb developers, business is not always growing. This can be due for instance to the fact that the targeted market does not react positively to the solutions proposed, or because your sales and negotiation power is sub-standard. The only consequence, which is heart breaking but not that harmful for the company, will be that you'll have to release part of your staff. They will have no difficulty whatsoever to find another job, given their abilities. Their time spent in the company will have drastically

risen their abilities, because brilliant people learn more when they are among other brilliant people. You must however be careful that the best do not leave while the mediocre remain. Or do not hire any mediocre staff...

In any case, given everything which was said about recruiting smart people, it becomes clear your staff is an important part of your capital. Many corporations state this only with a tongue in the cheek. They should be convinced of that, because smart people cannot easily be deceived! If the staff is your capital, you have to take care of it, for the company's benefit, which happens to be the staff's benefit too. For instance, excellent training must be provided. And the very same rule to avoid focusing primarily on salaries, but rather on productivity, applies again on trainers. The productivity of a trainer is even more difficult to assess during a training than productivity of a developer during a project. But it is obvious that extraordinary discrepancies exist between one trainer and another one. Usually, the best trainer is the most cost effective. As a consequence, looking for the very best trainers should become an obsession for the company.

Knowing that your staff is your capital should also push to establish an unlimited budget for books. Of course, this may scare your finance department, but from experience, I know that, in practice, this is not a source of huge expenditures. And a few ten dollars do not matter when compared to a few hours lost because the information was not readily available. Books must be in unlimited supply, not only to be useful within the technical scope of the day to day activities, but they should be accessible even for the sake of developers' general technical culture. We have to remember that the drastic reduction of trades turns developers into personnel who must have a broad technological culture. Allowing them to read books on technologies which are not contemplated for immediate use is precious.

All this contributes also to staff retention. On top of all this, reward must be fair. To many people, who have a "trade" view of developers and ignore productivity discrepancies, a developer position is a job which is "filled" or "not filled" by individuals,

in a binary logic manner. This is an enormous mistake and does not lead to a fair rewarding structure. The productivity discrepancies, which are used, must be acknowledged by the company, in some way or another. This means adapting bonuses in a way which allows discrepancies, and allowing for career development based on the real potential of the individuals.

1.13.4 Elite as a synonym to hacker

The word "hacker" did not initially refer to any truly criminal activity, only a very fuzzy sense of rules, combined with a technical passion justifying it. The word became gradually used for a very wide range of people, including some belonging to the extreme end of the spectrum, with criminal intents. Nowadays, hacking is fully equated to criminal activity in the public's mind. At the same time, as an inheritance of the past, hackers are often referred to as *geniuses*. The perverted fascination the public has for hackers and the publications related to them [32, 38] has placed them as modern dark idols.

Even among organized hacker groups, drastic selection was commonplace, as Bill Landreth [32] reports. The *Inner Circle* is one of the cyber criminal groups which reached fame.

Referring to oneself as a member of an *élite*, as a hacker, is now part of common hackers' slang probably since the 80s, in bulletin board systems. Between hackers, the word *élite* is often ciphered, and replaced by: Eleet, leetspeak, leetspeek, l33t, 31337, or 1337. 1337 is even used as a TCP/IP port number in a number of pieces of software.

This association between very smart people and piracy does not contribute favorably to the idea of forming serious industrial teams with very talented people.

On the other hand, hackers, sometimes for selfish reasons without any relation to philanthropy, have contributed to tremendous software breakthroughs. One of their main feats is the creation of the time sharing concept (Bob Bemer, also known as the inventor of ASCII) which is used by everybody now. Steven Levy explains it at length [33]. Similarly, a continuous chain of

innovation can be identified between the hacking of phone lines by Capt'n Crunch, the blue boxes allowing this hack, the electronic setup Steve Wozniak invented for them, and the video circuitry timers that he inserted in the Apple II.

Hence, hackers should not be considered merely as immature criminals, and smart people are not always cybercriminals.

1.13.5 Anti-intellectualism

Anti-intellectualism in the American psyche was pinpointed in the sixties by the American historian Richard Hofstadter, in the now classic "Anti-Intellectualism in American Life" [25]. He won the Pulitzer Prize for the combination of this and a former opus, "The Age of Reform". Hofstadter remarked that practical, pragmatic and functional knowledge were praised over intellectual and abstract abilities, leading to the ideal of the "common man" versus an educated elite.

Anti-intellectualism is nothing new, and tracks of it can be found many centuries ago. Yingzheng of Qin, the first emperor who united China, who came to the throne in 246BC, burnt books and buried intellectuals alive as he considered they were a potential threat to his administration.

In 1642, the New England puritan writer John Cotton wrote "The more learned and witty you bee, the more fit to act for Satan will you bee.".

Anti-intellectualism has increased during the last third of the twentieth century in America, and to a lesser extent in other parts of the world.

An interesting analysis can be made on the acceptation of the word "elitism". Up to the last third of the twentieth century, elitism was a positive term, sometimes associated with the word "republican" (republican elitism). During the last third of the twentieth century, it became pejorative. Few people remember that elitism related to rewarding merit, and was an idea pushed by revolutionaries. This concept became ultimately a republican value. It was one of the aspects of "equality" (implicitely of chances).

The French expression "la carrière ouverte aux talents" (career open to those with talents), brought by French revolution, was fighting the idea that some positions would be obtained with money, network or simply because of being born in an aristocratic family. Before the revolution, to become corvette captain, one had to belong to established nobility. The revolutionary expression was later echoed by Napoléon Bonaparte.

Anti-intellectualism is actually both a threat and an opportunity for the software industry. Randall Stross [51] elaborates on the negative side of it, stating that the level of hatred that Microsoft has generated towards itself owes a lot to anti-intellectualism. It must be underlined that it has also been Microsoft's opportunity, because not so many recruiters are tempted to recruit the top 5%. Surprisingly, there was never fierce competition to attract them, who are by definition in very limited supply. This does not only hold for the United States, but for Europe as well, in spite of a lower level of anti-intellectualism.

To be perfectly fair, among people having a very strong affinity with abstract matters, a lot of variety can be found. To some extent, the education system which imposes a pitiless selection on individuals, does not even fully satisfy the needs of the software industry. The nature of software development, with the analytical skills and mathematical rigour it requires, pushes for recruiting among people who have made brilliant scientific studies, especially in mathematics. Unfortunately, scientific studies usually do not select people on their creativity. Rather, they are selected on the ability to:

- Resolve a tough problem which is exhaustively formulated for them, all input being necessary;
- Work by relying only on their memory and often without any documentation;
- Manage their task by themselves;
- Perform in a very limited time, which is dictated to them;

Actually, the *real* world of the software industry is obeying the following, very different rules:

- The problem is never formulated completely and the people are *asked* to contribute to the formulation before resolving it. The problems to resolve are often incomplete, and the information gathered often contradictory.
- All documentation available can usually be obtained.
- Most of the tasks are done in groups, and the ability to be accepted and fruitful as a team member is key.
- You are consulted to evaluate the amount of time necessary.

More generally, students are taught to identify faster and faster patterns they have been presented during formal lectures. Pattern matching and creativity are two complementary abilities, both equally interesting in the software business.

The gap between the actual needs of the software industry and the educational system, which is, in this instance, more important in Europe than in the United States, could be partly resolved, only by "changing the rules" very slightly. For example, by breaking the sacred law of science education and problem solving, and proposing problems with extra unneeded data, it would be far more difficult for students. It is absolutely obvious that with this very tiny change, the ranks of the students would change meaningfully.

A funny example, used as a riddle for kids is the following: "Three crows are on a branch, a hunter comes and shoots one, how many crows left on the branch?". Such a question is often answered "2", instead of 0. Even, some science students, eager to answer fast, and influenced by years of study, would certainly answer 2. The fact is that the number 3 in the riddle is not necessary, it could be 5 or 7. When the hunter shoots, the remaining crows all fly away... Even the number of crows killed is not necessary to solve the riddle. The hunter could miss its target, no crows would be left anyway. Unconsciously, we all try

to find a use for the numbers 3 and 1 in the riddle. The only sensible use we find is to decrement the number 3. This is even more the case after years of practicing theoretical problems and having one's brain "shaped" by these problems.

Many riddles of the same kind as the "crow riddle" exist, which are much more sophisticated, with trains, speeds etc. Many are even used by Microsoft as a tool to select candidates. This practice is useful to compensate the deficiency of the education system in selecting individuals on their creativity. Real life is closer to the "crow riddle" than they are to math problems.

Having said that, there are many kinds of brilliant people. Some would be more inclined to spend 20 years trying to prove one of Fermat's conjectures. While this is perfectly respectable, and even admirable when it succeeds (cf. the marvellous work of Andrew Wiles), this inclination is in no way compatible with the requirements of software production. Brilliant people must also clearly be very practical, and willing to deliver from a quantitative standpoint, not just theoretical. Xerox Parc is an example of an organization which greatly contributed to computing, but globally in a very inefficient, and frequently too theoretical way.

It is important to note as well that being practical is also not sufficient in itself...

1.13.6 The moral issue

As a general and inexorable trend, the industry is moving to intellectual services, away from the Industrial Age and classical manufacturing dominated by trades. The fact that inter personal productivity ratios vary so greatly among all intellectual activities, indicates too clearly that the inequalities of rewards will owe a lot to natural abilities rather than merit. Additionally, software development is such a revolutionary activity that it resembles a tornado. It breaks organizations, reshuffles them, and has no pity for established practices. This is both beneficial to the businesses through increased automation, and detrimental to many people for putting hoards of them out of a job.

Believe it or not, there are examples of user companies,

which, after a successful implementation of a software solution, do not readapt their organization, and keep some of their elder staff doing their previous task, without telling them that this task has been automated, and that, after their intervention, the system will automatically redo it all.

Revolution is so intrinsically related to software development that it even applies in the most acute manner, to the software industry itself. Thielen [56] quotes Chris Peters, the former head of Excel development at Microsoft:

> We didn't write Excel to make money, we wrote it for the sheer joy of putting the largest computer software company out of business.

This statement is rather faithful to the general mindset in which successful software companies are, for excellent reasons, but is also absolutely shocking.

This mix between necessity and ethical issues is not new. The division of labor satisfied Adam Smith's moral system, because it was a way to offer work to anybody ready to work hard. It maximized the ability of a society to employ everybody, showing that the division of labor reflected God's will, who created the world with a role to play for all souls. In that sense, Smith, professor of moral philosophy at the University of Glasgow, was a Christian Humanist. The Christian Humanist school of thought lasted to this day. Peter Drucker (1909 — 2005), for instance, was also a Christian Humanist, and he is known, among several contributions, as the inventor of management by objective, and as the one who coined in the expression "knowledge worker". The refusal to admit or report that modern companies merely turn intelligence into profit (software or other technologies being secondary to that higher principle), testifies of his belonging to the long line of Christian thinkers. We should resist the temptation to see hypocrisy in the expression "knowledge worker" which presents mankind as a set of (implicitely more or less equivalent) empty bottles ready to be filled with information, and which, once full, will be available to regurgitate it. Drucker's generous ideology prevents him to see

the harsher nature of dissimilarity between employees, and the fact that knowledge without intelligence, like raw data without processing, is worth exactly nothing today.

> The same thing may be said of the gross ignorance and stupidity which, in a civilized society, seem so frequently to benumb the understandings of all the inferior ranks of people. A man without the proper use of the intellectual faculties of a man, is, if possible, more contemptible than even a coward, and seems to be mutilated and deformed in a still more essential part of the character of human nature. Though the state was to derive no advantage from the instruction of the inferior ranks of people, it would still deserve its attention that they should not be altogether uninstructed. The state, however, derives no inconsiderable advantage from their instruction. The more they are instructed, the less liable they are to the delusions of enthusiasm and superstition, which, among ignorant nations frequently occasion the most dreadful disorders. An instructed and intelligent people, besides, are always more decent and orderly than an ignorant and stupid one. They feel themselves, each individually, more respectable, and more likely to obtain the respect of their lawful superiors, and they are, therefore, more disposed to respect those superiors.

Who could have written such a cynical text, separating knowledge and intelligence, admitting inequality as fact, and assuming that there are higher and inferior ranks of people? Adam Smith himself, in the Wealth of Nations... The same one who wrote that there was little difference between a philosopher and a street porter.

The discrepancy between the generous theory and hard reality is very well known today in the case of manufacturing industries. Unfortunately, turning people into "instruments", as shown by Charlie Chaplin in "Modern Times", eventually raises

very significant ethical issues. Additionally, the division of labor gave birth to Taylorism, with an over zealous use of chronometers and productivity measurements, which literally killed a lot of people by putting too much pressure on them. Anyway, much before Taylor, Adam Smith was very well aware of this discrepancy, and, in his writings, criticized his own division of labour:

> In the progress of the division of labour, the employment of the far greater part of those who live by labour, that is, of the great body of the people, comes to be confined to a few very simple operations; frequently to one or two. But the understandings of the greater part of men are necessarily formed by their ordinary employments. The man whose whole life is spent in performing a few simple operations, of which the effects, too, are perhaps always the same, or very nearly the same, has no occasion to exert his understanding, or to exercise his invention, in finding out expedients for removing difficulties which never occur. He naturally loses, therefore, the habit of such exertion, and generally becomes as stupid and ignorant as it is possible for a human creature to become. The torpor of his mind renders him not only incapable of relishing or bearing a part in any rational conversation, but of conceiving any generous, noble, or tender sentiment, and consequently of forming any just judgment concerning many even of the ordinary duties of private life. Of the great and extensive interests of his country he is altogether incapable of judging; and unless very particular pains have been taken to render him otherwise, he is equally incapable of defending his country in war. The uniformity of his stationary life naturally corrupts the courage of his mind, and makes him regard, with abhorrence, the irregular, uncertain, and adventurous life of a soldier. It corrupts even the activity of his body, and renders him incapable of exerting his

> strength with vigour and perseverance in any other employment, than that to which he has been bred. His dexterity at his own particular trade seems, in this manner, to be acquired at the expense of his intellectual, social, and martial virtues. But in every improved and civilized society, this is the state into which the labouring poor, that is, the great body of the people, must necessarily fall, unless government takes some pains to prevent it.

Today, talking about recruiting the "top 5%" sounds unavoidably very close to a selection of "better people", reminding of dark times, Nazism, and their Lebensborns. The fact that we already entered the Information Age, and its combination with salutary competition, pushes corporations to be more efficient, and to fight for attracting the most productive people. This certainly raises a question mark on the ultimate shape of our society, and the social utility of this trend. The answer is far from obvious.

Chapter 2

Peculiar people for a peculiar world

Pick up that piece of paper Marvin,
that's what they say to me.
Here I am, brain the size of a planet,
and they ask me to pick up a piece of paper.

Robot Marvin,
in "The Restaurant at the End of the Universe",
by Douglas Adams

In the former chapter, I have extensively used vague expressions such as "smart", "superb", "brilliant", "high quality", "exceptional" to refer to the type of staff a software organization should recruit... What do I mean exactly? Are we talking about intelligence? IQ? Something else? The goal of this chapter is to shed some light on this.

We have to remember Knuth's mysterious comment of individual discrepancies in software development:

> I don't look at it as a matter of one person being better than another. Some people are simply going to be able to write much better code, but their code

> isn't necessarily going to be the better system for someone who doesn't think like the programmer.

To start with, we are going to study the author of these words, Donald *Ervin* Knuth, who happens to be the most typical, while certainly most extreme, of developers. We'll continue to clarify what was meant by "smart people" in the remainder of the chapter, and some risks related to them.

2.1 Donald *Ervin* Knuth

Donald Knuth is the undisputed greatest computer scientist alive, and one of the greatest ever, if not the greatest. He can arguably also be listed among the very best developers this world has known. He is a very interesting *case* for a study on the type of people software attracts and, certainly, shapes.

2.1.1 One hexadecimal dollar

Donald Knuth is, to computer science, the modern equivalent of Denis Diderot, the famous XVIIIth century Encyclopaedist. He holds a Ph.D in mathematics on the very casual and mundane subject of *Finite semifields and projective planes*. He is better known as the author of *The Art of Computer Programming*[1], the gigantic encyclopaedia on algorithms he started publishing in 1968 (217 years after Diderot...). He is still working on the much anticipated fourth volume. His contribution to software engineering through his books was such that he was appointed *Professor Emeritus of the Art of Computer Programming* at Stanford University, a title created exclusively for him. Somehow, he went much further than Diderot, and did not only collect algorithms created by others. He actually invented an incredible amount of algorithms. This is true to the point that all pieces of software available today, use, or required to be constructed, at least one algorithm invented by Knuth, or derived from one of them. Nothing less.

[1] Known as "TAOCP".

There are lots of passionating sides to Professor Knuth's life, but we are going to concentrate on what he calls himself his *peculiar* way of thinking. This way of thinking testifies in a very acute manner of the very special and queer twist of mind that excellent software developers have. This twist of mind is such that, to people who have not been exposed to software development, diving into it may feel like a very unsettling or even unpleasant, if not painful experience. Somehow, the amount of nausea you can feel measures how far you are from a developer's mind.

When he was a kid, Donald entered a competition which was organized by the confectionary manufacturer Ziegler. The goal was to devise how many words could be made with the letters of "Ziegler's Giant Bar". Knuth, at that time, was already a very bright schoolboy, and his peculiar spectrum of interest attracted him to this intellectual challenge. Pretending to be ill, he actively worked, and came up with 4500 words. Ziegler had only come to 2500, and he won the contest. His school received a brand new television set.

After starting his work on his gigantic encyclopaedia, he noticed how dissatisfied he was with the typesetting solutions he had access to. He quickly realized that the sophisticated kind of typesetting he expected was a job that could be endorsed by a computer, and that very interesting scientific issues could be derived from analyzing typesetting from an algorithmic angle. He therefore started working on a typesetting program that would become TEX. TEX is still one of the major pieces of software still being used for publishing technical reports, articles and books in the academic world. The document you are currently reading has been typeset using a superset of TEX called LATEX (for LAmport TEX, Leslie Lamport being the initial author of LATEX).

In order to understand the incredible level of detail reached by Knuth, a small parenthesis is necessary. First you need to learn what a "river" is. In documents typeset using classical text editors, paragraphs usually contain series of spaces which frequently, by chance, are vertically aligned, and look like small "rivers". You probably never noticed them. Now look at this

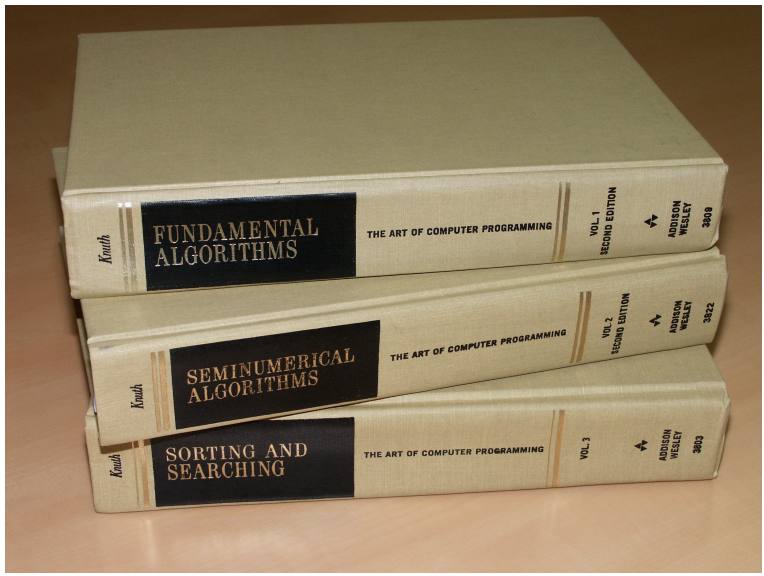

Figure 2.1: The Art of Computer Programming, volumes 1, 2 and 3.

text, you will not find any meaningful one. Knuth produced an algorithm allowing to avoid them. This tells you of a lot about the kind of details Knuth is concerned about. You will now notice rivers in deficient typesetting programs...

But the story is only beginning. Most of it is still to be told. Don Knuth actually went much further. In order to typeset a text, you need to display characters. These characters are drawn by a program, and the shape of these characters can be modeled through mathematical objects. He decided to go into further details, and design a program which would allow to draw fonts. He decided to use a scientific approach to that. His energy turned temporarily to another piece of software, necessary to produce the very high quality fonts he expected to deliver. This software was *MetaFont*. You certainly never wondered what family of curves could best allow to describe nice fonts? Knuth used a sub family of *Bézier curves*, namely the cubic ones. If

you look carefully at the characters used to print this text, they have been drawn using cubic Bézier curves.

Of course, with time, the shape taken by some symbols has evolved. The case of the Greek delta glyph is the most interesting. Two versions exist, the older one, and the final one.

Figure 2.2: The first version of the Greek letter delta.

Figure 2.3: The final version of the Greek letter delta.

Knuth dedicated an entire page on the web to the two versions of the delta glyph. With his usual mix of humorous and serious tone, he comments the situation as follows:

> For the Greek lowercase delta, you should tell your system administrator immediately to upgrade your obsolete version of the Computer Modern fonts.
>
> I made important corrections to all those fonts in the spring of 1992, but alas, I still see many books, journals, and preprints using the old versions. Please

> help me abolish the old forms from the typefaces of the earth.
>
> Many characters were improved in 1992 [...] but most of the changes are rather subtle compared to the dramatic improvement in the lowercase delta. In fact, the old delta was so ugly, I couldn't stand to write papers using that symbol; now I can't stand to read papers that still do use it.

When Knuth talks about abolishing the old form from the "typefaces of the earth", he is of course using a joking tone. But when he says that he could not stand to write papers using the former symbol, and that he cannot stand to read other people's papers using it, I am convinced he is *extremely* serious. Now, a proof that this document is well formatted using the proper delta glyph, defined with "perfect" cubic Bézier curves: δ. Or maybe you cannot make the difference? Don Knuth would.

When *MetaFont* and TEX were completed, Knuth was very concerned by possible bugs in his code. He made a formal proof of his code, and, as this was not sufficient, he published the following warning, formulated on a joking tone:

> Beware of bugs in the above code: I have only proved it correct, not tried it.

Knuth is always extremely serious and constantly joking. The first twist of mind is probably only sustainable with the second. As an early example of humor, Knuth published his first science article in a school magazine in 1957 under the title "Potrzebie System of Weights and Measures". In that article, he defined the fundamental unit of length as the thickness of MAD magazine #26, and named the fundamental unit of force "whatmeworry". This article was eventually bought by MAD magazine and published in the June 1957 issue.

Knuth is more famous for his $2.56 checks. He used to propose such a check for the first bug reports on TEX, during the first year. Year after year, he proposed to double the amount,

making a bet that he would be able to afford the exponential growth of the reward[2]. He extended the reward to errors found in *TAOCP*. It is a fact that very few checks were distributed, and on top, very few of them were cashed. They were rather framed or kept safely, as a bounty more valuable than the face value of the check. But, by the way, why $2.56 checks? Why this special amount? Simply because 256 cents is one hexadecimal dollar[3]...

Figure 2.4: One of Knuth $2.56 legendary checks. This one is for Yates *Arthur* Keir.

Now, back to inexorable precision, Knuth makes numerous references to other people's works in TAOCP. There is no unambiguous way to refer to a given person. Homonyms exist. He insists on mentioning not only the first and family names, but also the second name, at best, usually found as a single initial. Knuth has spent considerable efforts to find complete names for the people he refers to. A few of them escaped him, because they cannot be contacted anymore. This does not satisfy Don

[2]This exponential growth is a reference to the famous legend of the grain of wheat and the chess board. In this legend, a king promised to give as many grains of wheat as a chess board could contain, by starting with one grain on the first square, two on the second, and doubling at each square. This exponentially growing amount exceeded very fast the entire wealth of the country.

[3]One dollar is 100 cents and 100 in hexadecimal is 256.

Knuth. He has published the complete list of those "incomplete" people on his home page. He offers a $2.56 reward to anybody able to find a missing middle name, and to prove he has not made it up. And there are people insane enough to actually help Knuth find these missing names! I am one of them, happy enough to have obtained a $2.56 check from Don Knuth, after several hours spent on Google. A copy of my check can be found as an illustration. Unfortunately, when I sent the information to Don, he did not know anymore *why* he was looking for that person... He wrote to me that he would gladly give me another $2.56 check if I found *why* he was interested by that person. In spite of this, as a very serious person, he sent me the promised check... To date, I have not been able to find why Don Knuth was looking for that person, although I have a faint clue.

Now you understand why Knuth's middle name is typeset in italics in the title of this section... You never know, there could be a homonym for Don Knuth...

Now, professor Knuth considers his work on typesetting complete. He has published a set of books on the subject, the "Computers & Typesetting", published by Addison-Wesley [31]. But, the check business is not over. What happened in the unavoidable situations where the recipients of the checks did not receive them, and they were returned by the post office to Don Knuth? This situation is not one that would satisfy Don's peculiar way of thinking, as you guess. He has set a list of these people on his home page, and he asks the world community to help him find them, so that the check can properly be delivered. Guess what is the reward to help him? Do you believe he is joking? I can tell you he is not, you can try.

All of the above makes a lot of sense for a computer scientist or a developer. Having a "reject queue" is a very standard way to cope with errors. To somebody who has never been in the software business, it sounds like a very painful story. At this point, you realize now that "peculiar way of thinking" is only a mild description of Knuth's thinking process. But remember how much we owe all to Don Knuth... His twist of mind was absolutely necessary.

To be a little bit more complete, there is more. The following does not have anything to do with 1 hexadecimal dollar, but with the square root of 10!

2.1.2 3.16 or $\sqrt{10}$

We are far from having exhausted the Don Knuth "case", even from the "peculiar way of thinking" standpoint. This document will certainly not succeed in doing it. But a little bit more needs to be reported.

Don Knuth is professor of computer science at Stanford. When students were submitting the result of their work to him, it usually constituted a huge number of source codes, spread over numerous printed pages. There was no other way for Don Knuth, but to make a selection of a subset of those pages, and study them carefully, to come up with a reasonable opinion on the quality of the work, in a reasonable amount of time. Knuth chose these pages at random, knowing perfectly well that random algorithms are sometimes the only ones capable of resolving a problem efficiently.

He quickly applied the same to books selection. Don Knuth is an avid reader, and he was, like you or me, confronted with the problem of evaluating in a book shop if he would or not purchase a book. The only perfect way to know this is to actually *read* the book. But after, it is too late... Knuth is a very rational mind, and he came up with a solution to this problem. He would choose a page, at random (we shall see what this means in Knuth's mind), read it entirely, and figure out from this *sample* if he liked the book or not. Of course, the question is, what random function would he choose? He chose to systematically read page 316 (the joke is that 3.16 is square root of 10...). When a book would be shorter than 316 pages, he would read the page corresponding to the remainder of the division of 316 by the number of pages in the book.

In Knuth's life, computer science is certainly the dominant activity, but he has two other ones, playing pipe organ and bible study. Of course, the devouring passion for computer science

has deeply influenced him in other aspects of his life. The most peculiar work he ever produced is "3:16" [30]. "3:16" is a cross cut of the Bible, analyzing in depth, like with a magnifying glass, each 16^{th} verse of each 3^{rd} chapter of each book of the Bible...

Don Knuth made a series of 6 speeches on 3:16 at MIT. The records of these speeches can be obtained from Doctor Dobb's.

Now you understand what "peculiarity" means. In the MIT lectures, Don states he has a "peculiar way of thinking". I believe he means all the different ways to interprete "peculiar". There is of course the usual interpretation, which is synonymous to "queer", "strange", "eccentric", but also the more litterary meaning, which can consistently be found in the Bible itself. And Don Knuth knows the Bible well. Here is a first example of the word "peculiar" used in the Bible (Peter 2:9):

> Ye are a chosen generation, a royal priesthood, and holy nation, a peculiar people.

Or (Exodus 19:5):

> If ye will obey my voice indeed and keep my covenant, then ye shall be a peculiar treasure unto me above all people.

Or also (Titus 2:14):

> Who gave Himself for us that He might redeem us from all iniquity and purify unto Himself a peculiar people, zealous of good works.

In all these quotes, *peculiar* has a completely different meaning, obviously. It is both associated to a "privilege" (being owned by God) and a "responsibility" (being on Earth to do something about it). Knuth explains extensively in the 3:16 lectures that the incredible amount of efforts he spent during the last decades on TAOCP were felt as a responsibility to use his *peculiar* talent. So, when he talks about his "peculiarity" he means it from all the possible standpoints.

By the way, Don offers a check of $3.16 for any typo or mistake found in 3:16...

2.2 Richard Stallman

A chapter dedicated to exploring the *peculiarity* of excellent software developers would not be complete without a few words on Richard Stallman (also known as `rms`, for Richard Matthew Stallman). Richard is arguably one of the very best developers of all times. Here is a small list of the awards he received:

- 1990: MacArthur Foundation fellowship.
- 1991: The prestigious Grace Hopper Award from the Association for Computing Machinery (ACM), for his work on the Emacs editor.
- 1996: Honorary doctorate from the Royal Institute of Technology in Sweden.
- 1998: He shared with Linus Torvalds, the inventor of Linux, the Pioneer award of the Electronic Frontier Foundation for GNU/Linux.
- 1999: Yuri Rubinski Memorial Award.
- 2001: Second honorary doctorate, from the University of Glasgow.
- 2001: The Takeda Techno-Entrepreneurship Award.
- 2002: National Academy of Engineering membership.
- 2003: Honorary doctorate, from the Vrije Universiteit Brussel.
- 2004: Honorary doctorate, from the Universidad Nacional de Salta.
- 2004: Honorary professorship, from the Universidad Nacional de Ingeniería del Perú.
- 2007: Honorary professorship, from the Universidad Inca Garcilaso de la Vega.

- 2007: Honorary doctorate, from the Universidad de Los Angeles de Chimbote.
- 2007: Honorary doctorate, from the University of Pavia.
- 2009: Honorary doctorate, from Lakehead University.

His two main software "masterpieces" are `gcc`, one of the most popular compilers on the planet, `gdb`, its debugger, and `emacs`, a very famous text editor (and much more). He is better known now as the founder, in 1984, of both GNU and the free software movement[4] for which he is now one of the active evangelists. Richard practices self-derision about that and refers sometimes to himself as Saint IGNUcius, and likes to dress with a chasuble, while wearing a halo made of a hard disk platter. Here is a quote from Richard:

> People sometimes ask if St IGNUcius is wearing an old computer disk platter. That is no computer disk, that is my halo. But it was a disk platter in a former life. Unfortunately, no information is available about what kind of computer it came from or what data was stored on it. However, you can rest assured that no non-free software is accessible on it today.

You have to compare the self-derision Stallman demonstrates with the famous picture of Einstein, showing his tongue.

You may think that Richard is simply an easy going person, always ready to have fun. Richard is also a very active speaker, with a very tight schedule, and without any holidays ever. He is a missionary who travels extensively throughout the entire world to propagate the word of free software. He could certainly write a book with all the funny experiences he has had during his travels. Like most of the very successful software developers, he is very detail oriented, up to a point which may even worry you, or give you nausea, like in the previous section, if you are not a software developer.

[4] Free in the sense of freedom.

Here is an excerpt of a typical text (spanning many many pages) sent to his hosts when he comes to give a speech:

> A microphone is desirable if the room is large, but I have a very loud voice, so I don't need one for a small or medium room. A supply of tea with milk and sugar would be nice; otherwise, non-diet pepsi will do. (I dislike the taste of coke, and of all diet soda; also, there is an international boycott of the Coca Cola company for killing union organizers in Colombia). If it is good tea, I like it without milk and sugar. No other facilities are required.

Another one, showing his inclination for political matters:

> If you plan to restrict admission to my speech, or charge a fee for admission, please discuss this with me in advance and get my appoval for the plan. I'm not categorically against limiting admission or fees, but excluding people means the speech does less good, so I want to make sure that the limitations are as small as necessary. For instance, you can allow students and low-paid people and political activists to get in free, even if professionals have to pay. We will discuss what to do.

Richard is known to do his best to answer each and every mail he receives. Given his exposure, you guess that it means plenty of mail. Many years ago, I have personally experienced an extremely long exchange with him, and his availability and inclination for exchange of ideas is simply stunning. I was just an anonymous person emailing to him, and he took as much time as he could to discuss with me.

His inclination again for dealing with the tiniest details:

> If you are making a recording, please *make sure* to tell me when the tape needs to be changed. I will pause. Please help me help you make the recording complete.

Richard lives with a very little amount of money, far less than the minimum you would think of. Do not forget he is an evangelist, in the same spirit as the ones of the old times. Here is a little quote on plane tickets:

> Some organizations feel that hospitality calls for providing me with a business class ticket. That is indeed more comfortable, but an economy class seat is good enough even on an overnight flight if it is a window seat. Meanwhile, speaking is my main source of income, and the extra price of a business class ticket would be a lot more useful for me if I can spend it on something else. So if you were thinking of spending extra for business class, how about you pay the extra to me as a speaker's fee instead?

His obsession for accuracy can be testified by the following rules he sends to his hosts:

- He is okay with staying at a hotel, but he prefers a spare couch or just the floor. The floor is okay because he's always bringing an air mattress with him. But he needs a door for privacy.
- If the place is hot or humid, he needs air conditioning. At 72 fahrenheit and above he can't sleep. There is an exception to this, air conditioning is optional if temperature is above 72 and the air is dry.
- No cats, because he is allergic. Dogs that jump on him frighten him. The first exception to this is that it is okay if they are small. The second exception is if the dog is a "jumping kind" but does that only at first encounter.

And it continues on the specific and utterly serious problem of *beds*.

Unlike most, he doesn't like hard beds. But he is conscious of the fact that hardness assessment is a very subjective matter. So his host is encouraged to compare with the softness of futons.

Futons are too hard for him, so if your bed is significantly softer than a futon, it will be okay, otherwise it won't. Smart, when an absolute parameter cannot be assessed, there might still be a total order... And in the case of beds hardness, there is.

In case the first point fails to be properly assessed (if you have never heard of what a futon is, for instance...), you have to get a hair dryer ready for him to inflate his mattress. Oh, but there is another significant point to add, the dryer needs to have a "cool" setting, so that it doesn't damage the mattress. The instructions do not say what happens if beds are too hard and dryer does not have a "cool" setting. Shame.

Be reassured, Richard has a "loopback" mechanism, and he knows that all this might cause some laughter, and he is capable of self-derision. He finishes saying:

> In case you are wondering, I cannot feel a pea under a mattress, but I might feel a peanut under a thin mattress.

Now, what about food? First he explains he doesn't like anything spicy. But he becomes quickly very specific:

> Some foods I dislike include:
>
> - avocado
> - dessert that contains fruit or liqueur
> - fruits that are sour (grapefruit, most oranges)
> - hot pepper
> - liver (even in trace quantities)
> - stomach and intestine; other organ meats
>
> I often dislike foods that taste strongly of egg yolk, and some strong cheeses.

And on sightseeing:

> I enjoy natural beauty such as mountains and rocky coasts, ancient buildings, impressive and unusual modern buildings, and trains. I like caves, and if there is a chance to go caving I would enjoy that. (I am just a novice as a caver.) I often find museums interesting.
>
> If there is a chance to watch folk dancing, I would probably enjoy that. I tend to like music that has a feeling of dance in it, but I sometimes like other kinds too. However, I generally dislike the various genres that are popular in the US, such as rock, country, rap, reggae, techno, and composed American "folk". Please tell me what unusual music and dance forms are present; I can tell you if I am interested.

If Richard Stallman did not exist, we would have to invent him.

Now, as a child, what was Richard like? You guess he was a child prodigy. Richard was eight years old, when, as his mother reports, he solved a tricky problem published in Scientific American, to tease the readers. He would have to be called 9 or 10 times for dinner before he would actually come, because he was so absorbed by what he was doing. After a while, as Richard says himself: "After a certain age, the only friends I had were teachers". Dan Chess, now professor of mathematics at Hunter College, used to be one of Richard's fellow classmates. He says about him: "He was also smart as shit. I've known a lot of smart people, but I think he was the smartest person I've ever known". His "intensity" was remarkable, and he was very hard-headed too.

Sam Williams, the author of "Free as in Freedom"[5] [60], compares some of Richard's psychological traits to the Asperger Syndrome, also known as high-functioning autism. Stallman himself admitted, during an interview that he was "borderline autistic". This may explain his obsessive affinity for software development and computers.

[5] http://www.faifzilla.org

Being *borderline*, and staying always on the safe side while dangerously playing with coming closer to the pathological side, is certainly a feeling that most software developers have experienced to some extent. Self-derision, of the kind practiced by Knuth or Stallman helps keeping a link with reality.

2.3 Peculiar Jokes

We just saw that peculiar people need to keep a good sense of humor in order to remain on the safe side. It sounds natural that peculiar people make peculiar jokes... Let us have a look at two typical jokes which are particularly funny for developers, and only raises the eyebrows of regular mortals.

The first one is from Knuth. It is a joke he made during his 3:16 speeches at MIT. 3:16 is the weirdest topic you may ever think of. And it is a vast one. Don cut his speech in six chapters, and distilled it by fractions to the audience. In his introductory speech, he tried to explain why he cut it in 6. His answer is that he could only think of 6 jokes. Pretty dumb explanation, and lousy joke. But we are still on ground, the next part of the joke is stratospheric, as he adds: "and this was the first one...". Huh? Yes, follow him: he only thought of six jokes, therefore he wanted to split his speech in six, so that each of the parts had at least one joke. But saying it, is a joke in itself! So it would make seven... No, because it is the first one, the one he meant for the first speech. This is hilarious for a developer, and completely lousy for a mere mortal. Why so? Simply because it is a looping reference, one sentence in the set of jokes refers to the set it belongs to, and therefore becomes a joke itself. In other words, if you have `n-1` jokes, you can have an extra one, to reach `n` jokes by simply stating that you devised `n-1` jokes, and this, in itself, is an additional joke. Well... It turns out all this holds some similarity with the original works of the mathematician Georg Cantor, which later inspired Bertrand Russell, Kurt Gödel and... Alan Turing. In the same spirit, in a lecture given by Gérard Berry at the Collège de France, Gérard jokingly (is it really so?) proposed to

support Alan Turing as a recipient of the... Turing Award, the "Nobel" in Computer Science, as a homage to Turing's affinity with diagonal arguments. All these jokes, intertwined with hard science, say a lot about the ones who tell them or on the ones who laugh at them.

The other one, which is even more typical of a developer's peculiar sense of humor is the name Richard Stallman gave to his movement. Richard Stallman wanted, through his free software movement GNU, to fight against commercial brands, and to produce an operating system which would not be proprietary. His enemy was Unix, which was a trade mark of AT&T. What is the meaning of GNU? It is an acronym, which means "Gnu is Not Unix". Aha, but this is a *recursive* acronym! Recursivity is a very hilarious concept for developers... It has been reused to give a name to *PHP* which means "PHP: Hypertext Preprocessor". At some point in time, more hilarious names were given, in relationship with `emacs`, one of Stallman's babies: EINE meant EINE Is Not Emacs, and ZWEI: ZWEI Was EINE Initially — even better. In popular culture, the closest joke we can find is "there are two kinds of people, the ones who say that there are two kinds of people, and the others". Not everybody finds it funny...

We were talking about the Asperger Syndrome in the former section. Without assimilating developers to autistic people, there is a similarity with the perception of what is a joke. An anecdote is reported by Oliver Sacks in "The man who mistook his wife for a hat" [47]. He was studying two autistic twins who had fun together, saying something briefly to each other triggering an explosion of laughter right after. He discovered they were throwing prime numbers at each other. This was sheer joy for both of them.

2.4 Intellectual, but scientifically minded as well?

We identified in the former part of this document that the most productive people in software development were people capable of very high intellectual productivity. "Intellectual" is very vague, what does it mean in this case? Let us consider a few examples to draw some conclusions.

We know that Donald Knuth received a Ph.D. in mathematics on the subject of "Finite semifields and projective planes", from the California Institute of Technology. What about others? Here is a list, sorted by family name, of a few successful developers or IT specialists. Let us see if they have made bright scientific studies:

- Brian Behlendorf, the creator of the famous web server Apache, studied at the University of California-Berkeley.
- Sergey Brin, co-founder of Google, received a Master's degree from Stanford University, but did not complete his Ph.D.
- Dan Bricklin, author of the first spreadsheet, Visicalc, holds a B.S. in Electrical Engineering/Computer Science from MIT and an MBA from Harvard University.
- David Filo, co-founder of Yahoo, received a Master's degree from Stanford University.
- Bill Gates, studied at Harvard between 1973 and 1975, but never completed his studies because of the expansion of Microsoft business.
- Adele Goldberg, who greatly contributed to the creation of SmallTalk, received a Ph.D. in information science from the University of Chicago.
- James Gosling, the inventor of Java, one of the two most successful programming languages nowadays, holds a Ph.D.

in maths from Carnegie Mellon University for a work on "The Algebraic Manipulation of Constraints".

- Grace Murray Hopper, one of the very first programmers, received a Ph.D. in mathematics from Yale.

- William N. Joy, co-founder of Sun Microsystems, and author of `csh` and `vi`, received a Master's of Science in Electrical Engineering and Computer Science from the University of California, Berkeley.

- Mitch Kapor, founder of Lotus Development and author of Lotus 1-2-3, received a B.A. and studied Cybernetics at Yale.

- Brian Kernighan, one of the inventors of AWK, received a Ph.D. in electrical engineering from Princeton University.

- Leslie Lamport, author of LaTeX, received a B.S from the Massachusetts Institute of Technology in mathematics. He also holds M.A and Ph.D. degrees from Brandeis University, also in mathematics.

- Larry Page, co-founder of Google, received a Masters from Stanford University.

- Dennis Ritchie, inventor of the C programming language, graduated from Harvard with degrees in physics and applied mathematics.

- Richard Stallman, the founder of GNU, got a *magna cum laude* degree in physics from Harvard.

- Bjarne Stroustrup, the inventor of C++, one of the two most successful programming languages nowadays, holds a Ph.D. in Computer Science from Cambridge University.

- Ken Thompson, famous for his contribution to Unix, received a Master's degree in electrical engineering, from the University of California, Berkeley.

- Linus Torvalds, the author of Linux, holds a Masters degree in Computer Science from the University of Helsinki, the best university in Finland.

- John Warnock, the creator of PostScript, a format widely used for laser printing, holds a Ph.D. in Electrical Engineering from the University of Utah.

- Steve Wozniak, co-founder of Apple, was bored at school because he was too bright. In 1975, he studied at the University of California, Berkeley and left to roll out the famous Apple I.

- Jerry Yang, co-founder of Yahoo, holds a Master of Science degree in electrical engineering from Stanford University.

The list could go on and on. It is interesting to note the amount of math backgrounds among the people mentioned, in comparison with other types of scientific backgrounds.

By the way, Edsger Dijkstra said:

> Besides a mathematical inclination, an exceptionally good mastery of one's native tongue is the most vital asset of a competent programmer.

Similarly, if we forget individuals and focus only on nations, we may wonder if there is not a correlation between excellence in software and excellence of the scientific education system. Let us take the laureates for the Fields Medal, the most prestigious reward for works in mathematics. If we look at all the laureates since the creation of Fields Medal in 1936, and count only the nations having had more than one laureate, and order them by decreasing amount of laureates, we get:

Country	Amount of laureates
United States	12
France	7
United Kingdom	6
Russia	6
Japan	3
Germany	2

This list does not exactly correspond to the hall of fame of the worldwide software industry, simply because maths and software are not exactly the same thing. Also, the numbers above are not that big, and statistics cannot be built on small numbers. But the list is close enough to be striking. We do not find any country without any meaningful software activity, when actually those are very numerous. We do not miss either meaningful countries building software products (considering that India and China are today essentially producing custom software).

Does the evidence above imply that *all* people performing superbly in IT have made brilliant scientific studies?

Does this imply that *all* people who made brilliant scientific studies will perform superbly in IT?

The answer is clearly no to both questions. The first one is contradicted by the vast number of people who have done reasonably good studies, and who succeeded in software. After all, the simple fact that the education system makes a selection based on the performance of a given person at a given time of his/her life, shows that it is not a perfect "filter", even if success in the studies was the only criterion. During life, people have different maturity levels, or can face various material or affective problems. This may handicap making excellent studies, but does not affect their potential.

Now, among the ones who made top scientific studies, as explained earlier in the document, a meaningful amount do not fit well the concrete requirements of software production. Software projects require a steady constant concrete production, not the invention of a very innovative solution to a punctual very difficult problem. This is the difference between a theoretical approach

and a concrete one.

2.5 The ellipse and the circle

Here is a little anecdote explaining the difference between a theoretical approach and a practical one. This anecdote revolves around Philippe Kahn, the founder of Borland. Philippe Kahn used to be professor of mathematics in Grenoble, France. He started becoming fascinated by software development, and decided to move away from teaching to start a development career. The story says that he applied for a one year course on compilation techniques that was organized by the University of Nice, on the French Riviera. His application was apparently rejected by the professor responsible for the cursus, and he was firmly encouraged to return to teaching mathematics, as his potential in IT was considered very weak. At the light of the enormous success of Borland and its reputation for producing lightning fast, superior quality compilers, this story is one of the funniest in the software history. It also highlights the difficulty to evaluate somebody through a mere interview or motivation letter.

When Borland released its first C++ development environment, Philippe Kahn was extremely proud to have been faster than Microsoft. As a — most likely intentional — ironic gesture, he decided to pick the French Riviera as one of the very first spots where he would give a presentation of his new baby. I was lucky enough to be part of the crowd who listened to Philippe Kahn, and witnessed the demonstration of the long awaited Turbo C++. The crowd was made up of a variety of people, from the industry, education and research.

Philippe made a very enthusiastic presentation, and did more than a bit of programming on the spot to show Turbo C++ capabilities. He developed a few pieces of code, compiled then, ran them. At the end of the session, the people sitting in the amphitheater were allowed to ask questions. One of them triggered one of the most interesting exchanges I have had the opportunity to witness. To me, it is part of computing history, in spite of the brevity of the exchange.

One of the people in the audience was a researcher. During the programming part of Philippe Kahn's "show", this person was utterly shocked by what Philippe had done, and the way he had programmed his small application. In order to understand what is behind the last part of the anecdote, it is necessary to explain the context. Turbo C++ is the implementation of a development environment for C++. C++ is an object-oriented programming language, invented by Bjarne Stroustrup during the eighties. Object-oriented languages have started to become very popular, something like ten years ago, because they constitute a meaningful progress, in many respects, to produce software. One of their interesting features is the ability to facilitate reuse of existing code. Through reuse, a software organization is capable of drastically cutting down production costs. Instead of reinventing the wheel, object-oriented programming languages allow you to define what a wheel is, and facilitate the incorporation of a wheel in other people's solutions. This capability is of course not new, and did not wait until object-oriented languages appeared to be exploited through classical languages. But object-oriented programming languages, in a nutshell, make it far simpler.

As the borderline between science and software is always pretty fuzzy, there is always a temptation to see software under the pure theoretical standpoint. Object-oriented programming, including the one made possible by C++, is no exception. The terminology used in object-oriented programming does not really help in considering it as a mere software tool, meant to facilitate developers' lives. Terms like "inheritance", and especially "polymorphism", contribute to the illusion that practitioners are scientists. Philippe Kahn, during his presentation on Turbo C++ made some fun out of this terminology, and ventured on the anti-intellectualism terrain his audience ordinarily expects from him. In this case, this was probably not well received.

At the end of his presentation, Philippe proposed the usual questions and answers session. One of the people among the assistance raised his hand. Instead of asking the question you would expect during such a presentation by a CEO of a leading

software editor, he asked a very detailed oriented question on the actual *piece of code* that Philippe had developed during the demonstration. The question was the following: *"I have a hard time understanding why you programmed the example the way you did. I always do it exactly the other way round. If I face the same problem, instead of having the ellipse deriving from the circle, I have the circle deriving from the ellipse"*.

This requires a decoder, a babel fish as Douglas Adams would say, to understand object-oriented jargon. As a matter of fact, Philippe Kahn understood the question very well, of course, and answered straight away, without giving the answer more than a few milliseconds of thought: *"That is exactly the difference between research and industry"*, and he proceeded abruptly with other questions.

You could believe that he did that to avoid having to answer a very difficult or tricky question. In fact, he could not give a better answer to the question. Elaborating more requires far bigger efforts, and Philippe's answer is an utterly striking summary of the *actual* answer. The most surprising is that it did not take him any measurable amount of time to figure it out. It tells a lot about the amount of *maturity*, and deep understanding of software development somebody like Philippe Kahn had reached.

Now, some explanations are required to understand what Philippe meant. Object-oriented development can either be seen as proposing the ultimate data modelling technique, that is an equivalent to a mathematics concept, or it can be seen as a mere, although sophisticated, instrument, to bring more comfort to developers. Philippe Kahn obviously belonged to the second category. Philippe had used, in his code sample, a piece of code, where the `Ellipse` class inherits from the `Circle` class. You have to think about a class as a description of a common profile that a number of objects (rather intellectual objects, actually) *share*. For instance, the *Circle* class can be fully defined by a point, being the center of the circle, and a radius. All circles have all these characteristics. Similarly, an *Ellipse*, can be fully defined for instance by a *center*, a *major axis* and a *minor axis*.

It is important to note that a circle is a particular case of ellipse, where the major and minor axis are equal to the diameter of the circle.

As a consequence, as a circle is an ellipse, it is fully satisfying, from a mathematical standpoint, to have a circle deriving from an ellipse in an object-oriented programming language. On the other hand, the inheritance rules in C++ impose that, as the circle is an ellipse, it has independent axis. As a consequence, the circle is unnecessarily storing two pieces of information, when it requires only one, the radius.

The confusion comes from the fact that inheritance, in C++, is not allowing to describe mathematical concepts, but it allows to *program*. Wanting absolutely to see classical mathematics or *ultimate* modelling means, in C++ does not mean much. Being fully satisfied from a mathematical or merely "set logics" standpoint does not mean that the programming language is used correctly. There is a huge gap between a theoretical approach and a practical software development approach, one that justified the disdain that Philippe Kahn exhibited when answering the question. In the case above, having a theoretical approach leads to producing unnecessary fat objects. The precious and unfortunately now out of print "Taligent Guide to Designing Programs" [52] calls this "anti-pattern" or "pitfall", *overeducated base classes*, and rightfully warns developers against such practices. Ironically, the *overeducated base classes* syndrom is often a trap for *overeducated people*!

Everything would of course be too easy if software development had nothing to do with science. Unfortunately it has... Edsger Dijkstra, famous for his paper "Goto Considered Harmful", and winner of the ACM Turing Award, wrote that "Programming is one of the most difficult branches of applied mathematics; the poorer mathematicians had better remain pure mathematicians." and "The use of anthropomorphic terminology when dealing with computing systems is a symptom of professional immaturity".

Now, was Philippe Kahn right? The real answer is that he wasn't. But he did not care. In C++, the ellipse should not

derive from the circle, and the circle should not derive from the ellipse (at least not naively). But we would enter too long a debate to explain that. Also, while all this holds true for C++, it may not hold true for all programming language. But one conclusion remains: software tools are only tools, and pure theoretical standpoints are dangerous in software development. It is a fact that there is some level of contradiction between making bright scientific studies, and having the *practical* sense software development requires. To some extent, the class of people who have too theoretical an approach for software development fuel the anti-intellectualism (cf. section 1.13.5) software development suffers from.

2.6 Peculiarity, Aesthetics and Culture

It may seem that there is an incompatibility between being a bright developer and having some interest in cultural matters. A common feeling is that being bright in development involves so much dedication that being open to the rest of the world is reputedly impossible. This is not true for the very best developers, and strangely enough, many of them have a vivid interest in music, litterature and art in general. This is something you would not expect from a purely scientific mind. It seems to me more intense in the software engineering field than in other engineering activities. Anecdotes are numerous about links between software development and culture. Let us take two of them.

One of the software pioneers, Peter Samson, had built at the MIT a program on the TX-0 machine, that allowed the machine to play music. Although this, in itself, shows some interest for classical culture, the core of the anecdote lies further. Peter Samson was famous for his unwillingness to insert comments in his code. This is reminiscent of Evariste Gallois, the young mathematics prodigy, who invented group theory, before dying in a fatal duel. Evariste Gallois hated to elaborate on his work, and frequently skipped parts of the demonstrations, because he found them too obvious, rendering a few parts of his work still somewhat obscure nowadays. In Samson code, one could find

long series of machine instructions without any comment, except a mysterious one: `RIPJSB`, right beside the constant `1750`. After a lot of interrogations, one of the puzzled readers figured out that `1750` was the year Bach died, and understood that `RIPJSB` was an acronym meaning *Rest In Peace Johann Sebastian Bach*. This is probably the most unexpected cultural reference in a piece of software...

The second anecdote is buried even further in computers architecture. In a computer, each piece of accessible memory is given an address, which is a numeric value. These individual pieces of memory are called *bytes*[6]. This address is used to retrieve the information stored in memory. The numeric value constituting the address can be increased or decreased in order to span the addressable memory space. When storing a number in a computer's memory, if the number is too large to fit on a single byte, it needs to be stored on several ones. The "most significant" byte is the one storing the most significant value constituting the number. For instance, in the decimal system, the number 1234 has 1 as the *most significant* digit, and 4 as the *least significant* digit. This is the same in the binary world of computers. Two choices exist when designing a computer, the most significant byte can be stored at the lowest address, or the highest. In these cases, respectively, the *big endian* and *little endian* approaches are used. Some computers support both of these modes. Most developers are very familiar with those terms. Few know their origin. These expressions actually come from Jonathan Swift's most famous opus, *Gulliver's Travels*. In that work, Swift describes the Lilliputian king, who asks his subjects to break their eggs at the smallest end, and fights arise between the "Little Endians" and the "Big Endians".

As we already saw, Donald Knuth is Professor Emeritus of the *Art* of Computer Programming at Stanford, and the word *Art* has to be accepted with all its meanings. This is espe-

[6]The word *byte* is often mistaken with *octet*. A byte is an octet when the individual addressable memory stores exactly 8 bits. This is not always the case, although it is frequently true, and depends on the computer architecture.

cially obvious with his work on fonts and the use of Bézier curves. These curves were originally invented by Citroën (Paul de Casteljau) and Renault (by Pierre Etienne Bézier) to build beautiful car shapes. These curves extended to many industrial areas and are now synonymous with "beauty", like the legendary "golden number". Knuth's interest in aesthetics is also obvious when you consider his taste for "manual" calligraphy, as testified by Bible verses calligraphies in 3:16 [30].

Interestingly enough, Robert M. Pirsig, in ZAMM ("Zen and the *Art* of Motorcycle Maintenance: An Inquiry into Values") [40], associates also the word "Art" with an activity he describes as very similar to scientific work — motorcycle maintenance.

Brian Kernighan, the inventor of `awk`, co-author with Dennis Ritchie of the famous book on C programming language [27] is known for teaching a very original subject at Princeton: programming *style*. James Coplien, in "Advanced C++ Programming Styles and Idioms" [11] says that "style distinguishes excellence from accomplishment", and he applies it to software development.

All this means that aesthetics and development are not only compatible, but they are, to some extent, even difficult to dissociate. Of course, the closer you get to programming, the more different the aesthetics are from classical arts. But remains the fact that a clear sense of aesthetics is required to become a very bright developer. This is not new, and applies even to mathematics, and examples are numerous of arts "missing links", which establish bridges between various aesthetics. Johann Sebastian Bach, Kurt Gödel and Maurits Cornelis Escher are a few examples, all three strikingly compared by Douglas Hofstadter in "Gödel, Escher, Bach: An Eternal Golden Braid" [24]. Those three clearly are *peculiar* in their own domain.

2.7 Intellectuals and their weaknesses

Being an intellectual does not provide an immune system against all parasitic ideas which can pollute the rationality of our brain. Some time ago, I visited a religious organization — I should

say a sect. The tour around the campus was made by a person doing scientific research work in a major national research institute. Truly a very respectable person. She explained that their guru, the previous night, had killed during his sleep 10,000 lemurians who try every night in secret to conquer the earth. 10,000. Lemurians. The site was covered with giant statues, some representing angels with machine guns...

Newton (1642-1726) himself was an alchemist — well, everybody was at that time, except just a few like Athanasius Kircher (1601 or 1602-1680) and especially Werner Rolfinck (became professor at the University of Jena in 1629).

The 13^{th} of June, 1838, the great scientist François Arago predicted that the most sweating people would catch pleuresy when passing through a train a tunnel, because it would create too quick temperature changes.

It is well known that even the most brilliant minds can easily be influenced by the most silly ideas. All this is a matter of being particularly vulnerable on one aspect of one's personality, and the brain fuses... This can be a collective phenomenon, and to some extent, the fact that it is collective makes it suddenly attractive to less vulnerable people, as a snowball effect.

Without any doubt, the same happens in the field of software development. The following contains just a few anecdotes on the subject.

2.7.1 No computer can ever play chess

This anecdote is reported by Steven Levy in "Hackers: Heroes of the Computer Revolution" [33]. During the mid sixties, a Rand Corporation memo entitled "Alchemy and Artificial Intelligence" circulated. It was written by an academic named Herbert Dreyfus. Dreyfus compared the complexity of the human brain with a computer's circuitry and concluded that a computer would be incapable of matching the type of intelligence a human brain is capable of. He concluded for instance that "no computer program would be able to play a good enough game of chess to beat a ten-year old" (from Levy).

Obviously, the MIT hackers were deeply offended by Dreyfus' paper. One of them, Richard Greenblatt, designed a chess program, MacHack, and invited Dreyfus at the MIT to play it. Herbert Simon, one of the artificial intelligence pioneers, reports:

> [...] a real cliffhanger. It's two woodpushers [...] fighting each other [...] Dreyfus was being beaten fairly badly and then he found a move which could've captured the opponent's queen. And the only way the opponent could get out of this was to keep Dreyfus in check with his own queen until he could fork the queen and king and exchange them. And the program proceeded to do exactly that. As soon as it had done that, Dreyfus' game fell to pieces, and then it checkmated him right in the middle of the board.

2.7.2 We can do anything with COBOL

When I started contemplating having a "real" permanent job in the '80s, there used to be a clear distinction between business processing and scientific software development. I was very curious about this distinction, and had the very intense feeling that all this would fall within the general "IT" basket, with technologies converging to a similar point. If my analysis was wrong, my personal inclination, which was to develop scientific software, would keep me out of business processing software, and I would be forever full of questions on that field. Being young pushes you to adventure, and I decided to enter business processing software development at the beginning, in order to explore this world. The company I was working for was *the* major French software editor, one of the largest in Europe, and started to enter the American market, with solutions which ranged from pure 4GL development, to a complete range of full fledge Enterprise Resource Planning solutions. From classical programming (C, ADA, LISP, Prolog, etc.) on open systems, I switched to COBOL development on a variety of mainframes. I have always had a very vivid curiosity for everything which deals with programming, and I learnt this language quickly, with a reasonable

amount of pleasure. The company I worked for was extremely successful, and it was later acquired by IBM, as part of IBM Global Services.

I saw no sign of the convergence I was expecting, until a German company started communicating around ERP solutions which were available on open systems (Unixes). At last, this was the sign I was waiting for. I rushed into the office of the chief technology officer of the strategic branch I was working for. I will remember all my life the chat I had with him. I started talking about Unix very excitedly (remember, I was young), and about other programming languages. To my surprise, my interlocutor was completely closed to any move. He felt that the German competitor company was a dwarf, that it would collapse quickly, and that COBOL was a universal programming language, capable of producing everything, including what we would call now a "Computer Aided Software Engineering" tool, or an "Integrated Development Environment". I realized that the entire company was fanatized by COBOL, and that any move was considered the most silly thing to do. Of course, nowadays, this position sounds utterly outdated, but at that time, for many people, it made a lot of sense. It was a time when chief information officers working for banks or insurance companies were reporting on front page magazines that they had engineered their own development environment. Crazy times. The company I was working for was no exception and was convinced that anything could be done with COBOL. This faith in COBOL went up to the point where it turned them completely blind. I resigned immediately after. If anything could be done using COBOL, it would be without me.

The German company was SAP, and the company I was working for saw its software customer base disappear completely throughout the years, without any reaction. The company witnessed its software editor arm, half of the activity, shrinking into oblivion. The most brilliant engineers were working for them, and unfortunately, they had become completely insensitive to the outer world, as they were so obsessed by COBOL.

SAP is now the most successful ERP software company, by

very far. Nobody is capable of matching them.

2.7.3 We squeeze everything out of X-Window

One of the companies I worked for, later on, was the leading supplier of object-oriented components in the world. This company was a spin-off of the largest research institute in IT and automatics. Their core technology was LISP-based. LISP is one of the very early programming languages, designed at MIT, at the times of the pioneers. Its users community is very closed on itself, and has some of the traits of an ideology, to a much greater extent than the COBOL community I was referring to above. LISP is reputedly very slow, and thus, sometimes people refer to it jokingly as "the most intelligent way to misuse a computer" (I endanger my life by reporting this...), and when I considered working for this company, I had a few worries on the fact that I would have to use LISP all day long.

When I met the CEO, an extremely bright person, with a stunning vital energy, I asked him his opinion on LISP performance, and in particular on the capabilities offered by the graphical toolkit they were selling at that time, and which drove most of the company's sales. The CEO explained to me that their tool was squeezing all the power out of the graphical system it ran on. This system was X-Window, a popular graphical system on Unix/Linux and modern Mac OS. The CEO told me that they were capable of refreshing 360 objects per second. I was not a specialist of X-Window, and that sounded okay to me. That company had convinced all its customer base (in addition to itself), that their LISP toolkit was the best on the market, and that it could not be beaten. Similar stories were circulating on the relative performance of LISP versus other programming languages, by the way.

I joined this company, and a few months later, I heard the most astounding story (see page 34). A — very — bright developer had isolated himself during three months (one of the key assets of the company was to allow any senior developer to do that, without any reporting), to develop a personal project. Af-

ter three months, this developer had asked the CEO to come and see him. Very curious, the CEO came and the following happened: the developer said "do you see these small patterns on the screen? Look, I select them, and I move my mouse. Look, they follow the mouse in real time.". Each of these small patterns was actually an independent object on screen. There were hundreds of thousands of them. And they could move in real time, following the movements of the mouse. We were not talking about 360 objects refreshed per second, those were hundreds of thousands of objects moved in real time, without sign of any problem on the X-Window side. Within a second, the CEO (he was extremely bright, again) realized the fanatism he had been living in, and on top, the fanatism which had been communicated to the staff, and to the customers. He asked immediately: "but what technology are you using?". The developer answered "this is C++ code, no LISP". This was such an earthquake for the company, that the developer was prohibited from communicating on his work within the company. The CEO was scared that the employees would all leave the company at once. This was evidence. From 360 to maybe 4-500,000 objects refreshed per second, only a 1,000 or 1,500 ratio...

The company decided ultimately to stop developing LISP software, stopped its LISP compiler project, switched to other technologies, grew comfortably, and is in quite a healthy situation, more than a decade after these events. Just a few people left the company when LISP was abandoned.

It is funny to note that the debate on LISP and on its addictive properties already existed since the very beginning, within the MIT premises. Levy [33] recalls very strong words:

> One of the tasks Gosper considered impossible was a useful LISP on a PDP-6 — it might be nice as a symbol evaluator, but not to *do* anything. He considered it one of Minsky's follies that Greenblatt and the others had been tricked into implementing.

Note the world "follies"...

Lutz Prechelt [43] demonstrated very brilliantly that interpersonal variations have a far greater impact on productivity and solutions efficiency than technology itself. This explains the inability of many software companies to evaluate technology correctly. After all, when they attempt to evaluate another technology, they only evaluate the skills or the technological affinity (sometimes fanatism) of their staff.

The most sensible explanation I find to the unavoidable addiction to some specific technologies is due to their intrinsic complexity. When you master, after a great deal of efforts, a given technology, you are naturally not inclined to change, and you stick to it with all your energy. Additionally, the tools themselves shape and affect the mind, as explained by Edsger E. Dijkstra:

> The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

Also, remember that a computer is a universal machine (following the exact terms coined in by Alan Turing, one of the fathers of computing), and the temptation is strong to consider a specific technology as the universal silver bullet for all problems (see section 2.7.2 on this subject).

This addiction can be compensated by Functional Division of projects, as explained on page 64.

2.7.4 Time will come when computers will be fast enough

This section recounts an anecdote I have seen repeating itself so many times, under a variety of forms. This one is the most typical. Technological fanatism, and fanatism in general can be testified by the sudden absence of logic, or the sudden blindness on an obvious side of a problem. The story which follows relates to LISP. It is just accidental that the former paragraph is also about LISP. I do not have any specific strong feelings against LISP, but it is simply obvious to most of the software developers

that this technology has not spread to the extent its community expected it to reach. It remains on a niche market, which will certainly survive a few decades. After all, as my friend Jean-Paul Rigault says, "programming languages never die".

When I was doing my PhD thesis, using the very minimal, but however efficient, C language, I got a visit from a prominent figure of the leading research institute in IT and automatics. He paid a visit to do some evangelism on LISP, and push me to abandon C. What was his argument? He said that, yes, LISP was slow, but it was simply a matter of computing power. One needed only to wait a bit, and thanks to Moore's law[7], the time would come when LISP would be efficient enough to be widespread. Therefore, it was a big mistake to continue investing on C, and it was high time to switch to a more elaborate programming language.

I have heard this reasoning so many times that I do not even remember how many. I still hear it at the same frequency, as if the frequency was universal like Planck's constant. There is an obvious swindle in the idea itself, a swindle so obvious that respectable people should not propagate it. Of course, it is now even more obvious as LISP did not invade our familiar environment. It is not related to the nature of LISP, for instance. The question is not whether LISP is actually superior to C or not, if it allows to produce software at a sustainably cheaper price, or with a higher quality. The question is: what do we do with the extra power allowed by Moore's law? The answer is very simple. The power offered by Moore's law is feeding an inextinguishable monster. A monster which is never satisfied, which is always so hungry that nothing could feed it. This monster is what you expect from your computer. The hunger for functionalities is so huge, that whenever a more powerful processor is released, the

[7]Gordon E. Moore, one of the founders of Intel, made a prediction in 1965. This prediction is now universally known as "Moore's law". It states that computing power doubles, for the same price, every eighteen months. Since it was stated, it has been remarkably verified, in spite of its exponential nature. It was actually initially stated for 12 months, and Moore revised it at the end of the '70s and replaced the 12 months parameter by two years. It is usually taken now as 18 months.

entirety of this power is eaten up by additional software features.

So, as Edsger Dijkstra explained it in an ACM Turing Lecture in 1972 [15], we are not in a situation where the equipment is a "painfully pinching shoe" to be relaxed when more powerful machines would be available. And Dijkstra was right when he said:

> [...] in those days one often encountered the naive expectation that, once more powerful machines were available, programming would no longer by a problem, for then the struggle to push the machine to its limits would no longer be necessary and that was all what programming was about, wasn't it? But in the next decades something completely different happened: more powerful machines became available, not just an order of magnitude more powerful, even several orders of magnitude more powerful. But instead of finding ourselves in the state of eternal bliss of all programming problems solved, we found ourselves up to our necks in the software crisis! How come?

Take the LISP example. At the time of the anecdote, no widespread office product (text processor, spreadsheet) was coded in LISP. No widespread operating system, no widespread database system was coded in LISP. Had they been coded in LISP, they would have been extremely slow, and so resource hungry that working with them would have been barely acceptable. Only the very powerful emacs text editor, programmed in C, uses extensively a LISP dialect for its extensions. But it cannot match popular office text processors when it comes to editing sophisticated documents. Of course, the functionalities offered 15 years ago by text processors, spreadsheets, operating systems and databases, were inferior by far to the ones we are enjoying today. Today, maybe, on modern computers, the user experience we could have on solutions coded in LISP might compare to what people were experiencing with non LISP technologies in the past. But who would care about that? For instance would

you like to revert to windows, which draw only their outline when you move them on your screen? No, today you want to see the window moving in real time. You want to see the content moving. And you want them to be semi transparent too. You do not want to switch back to early versions of Windows or Macintosh OS. Seriously, there were hoards of people investing and working on LISP-based operating systems, to have "pure" LISP machines...

The pressure of the hunger for functionalities is of course not the only driver. Marginally, it is true that a little bit of the power is used to allow for a slightly better productivity for the developers who produced the solution. But when you are in a competitive situation, every possibility to offer more functionalities for the same price prevails. Otherwise, you are quickly out of business, and a competitor will propose more to your customers.

My interlocutor had simply overlooked the main parameter driving the ever growing power of processors. He overlooked it, because he was so addicted, so obsessed by his addiction, that he became completely blind. This mistake is very frequent, because if one striking major law of computing is to be quoted, most people mention Moore's Law. This exponential law is so exceptional, that few can imagine that there is an even stronger one: the inextinguishable hunger for functionalities. This law is stronger by far, as a matter of fact, it is computing's equivalent of the *general gas law* from physics. Even if the container grows exponentially, gases will occupy all the space. Its computing counterpart could be called the *general functionalities law*...

As a summary, the software industry obeys the following law:

$$CT = nF$$

Where C is the competitive pressure, T is the acceptable response time (a relatively stable parameter), and F is the amount of functionalities.

Software professionals who ignore this "functionality law", and select technologies not capable of delivering the proper amount of functionalities as driven by this law will become preys for

newcomers. Of course, response time for a given technology evolves with time, thanks to Moore's law, but competitive pressure evolves too, often exactly at the same speed.

So now, if you hear: use this, it is slow now, but one day, computers will be fast enough to... be very careful.

2.7.5 When one has a hammer, everything looks like a nail

While it makes sense from a scientific standpoint to push all theoretical frameworks to their limits, the gap with reality and concrete matters is very wide. Between computing envisioned by Alan Turing and the application of programming languages for running banks, insurances or even embedded systems which go to Mars, there is some difference of approach. Indeed, delivering solutions based on computing requires overcoming difficulties with the best possible means, while, for instance exploring the potential of an algorithm, or a programming language is already part of science. This is precisely what Philippe Kahn meant about the difference between research and industry (see section 2.5).

But it is easy to be trapped and mix the two. This holds especially for programming languages. Programming languages are supposed to be universal (remember about Turing's universal machine), and it is therefore very easy to fall into some level of fascination about their abilities, especially if you mastered them after some pain, and are willing to rest on your recent leap forward. Most of the programming languages invented since the fifties are able to describe the processes running in a bank, an insurance or an embedded system going to Mars. This is both a wonderful — and again fascinating —, property of computing, and a handicap to make proper choices, as being able to do something, and doing it well — with all the different meanings it can have — are two different things. A great deal of irony comes from the fact that the fascination is all the more potent, that the language theoretical framework is simple.

Fascination, combined to the joy of mastering a set of com-

plex concepts leads very easily to fanatism. Fights about languages have been very frequent since the fifties, they have even been called "language wars", where camps fight to defend a vision of the world which fits their standpoint.

There are striking and funny historical examples. It has been perfectly possible to build (supposedly) general purpose programming languages proposing a vision of the world where everything is a list (LISP), everything is a string (SNOBOL), everything is a stack (Forth), everything is a pattern (Prolog), everything is an object (Smalltalk, or Eiffel), to speak of the most famous ones. When object-oriented programming started its propagation beyond the limited circles where it was known, to become a popular technology, a similar phenomenon happened, and the public saw objects as the last marvel of the world, resolving magically all the problems we had faced since paleolithic. At that time, I remember people answering to each technical issue raised in project workshops "it's an object!" as if it was the ultimate solution to everything.

Bjarne Stroustrup has created with C++ an entire different animal, which inherits from structured programming, while mixing it with object-oriented and then generic programming models. The very practical, however impure result, has been criticized by purists (see section 2.8.1 dealing with purity) for accumulating different models rather than drawing a single one unifying them. Java, later, has attempted to do the same while offering simpler constructs. Bjarne's approach, which is driven by practical considerations, aims at providing different solutions for different classes of problems, and, specifically for a design ensuring that you do not have to pay for what you do not use.

Blindness and fascination for a programming languages has caused a litany of failures. We have already talked about the insistence on COBOL, triggering the demise of a very large European product company (section 2.7.2), but other examples are numerous. Similarly, Smalltalk has kept Xerox PARC[8] unable to create Apple, while Steve Jobs, after a visit to the PARC

[8]Palo Alto Research Centre

realized immediately that all user interfaces would eventually behave like the one invented by Xerox for the Alto, and created the Macintosh, selecting the then more adequate assembler to develop it. Himself, after leaving Apple, created NeXT Computer, and his staff insisted on basing the system entirely on Objective-C, a decision which turned out to reproduce the same mistake as the one they fixed by eliminating Smalltalk from the Xerox system they had used as an inspiration. Not everything is a nail...

2.8 Technology and Society

Fashion, politics, philosophy, art, are obviously influenced by the directions taken by our society, and by the evolution of thought. Software is apparently the most improbable area that could be influenced by external factors. It seems to be an isolated activity, to some extent, attracting people who hate socializing and remain on the margin of the rest of the world. This is not — completely — true.

We have already pointed out in section 1.3 the relationship between the counterculture of the sixties, the hippies movement and the revolutionary nature of software. It is a fact that the hippy community, of which Jack Kérouac [28] was the pope, used to consider technology as foul. This is a theme highlighted as well in Pirsig's ZAMM [40]. Pirsig published his book during the post hippy period (1974 exactly), but the action takes place in 1968. Pirsig explains his relationship with a couple of friends who have a very strong aversion to technical matters, an aversion he discovers, as an enlightment, by analyzing their weird reactions. The book is about the motorcycle rides he makes with them, and the constant surprise he has about their lack of interest in understanding how their motorcycle works and can be properly maintained:

> It occurred to me that maybe I was the odd one on the subject, but that was disposed of too. Most touring cyclists know how to keep their machines tuned.

> Car owners usually won't touch the engine, but every town of any size at all has a garage with expensive lifts, special tools and diagnostic equipment that the average owner can't afford.

and:

> I might have thought this was just a peculiar attitude of theirs about motorcycles but discovered later that it extended to other things.... Waiting for them to get going one morning in their kitchen I noticed the sink faucet was dripping and remembered that it was dripping the last time I was there before and that in fact it had been dripping as long as I could remember. I commented on it and John said he had tried to fix it with a new faucet washer but it hadn't worked. That was all he said. The presumption left was that that was the end of the matter. If you try to fix a faucet and your fixing doesn't work then it's just your lot to live with a dripping faucet.

Note the "odd" and the "peculiar" in the two former quotes. Actually, it turns out that Pirsig realizes that the peculiar one is himself. Most people hate technical matters, and have grown a defiance with respect to technology. The ones having no fear belong to a scarce breed.

Another quote, with a direct reference to programming:

> It's not the motorcycle maintenance, not the faucet. It's all of technology they can't take. And then all sorts of things started tumbling into place and I knew that was it. Sylvia's irritation at a friend who thought computer programming was "creative". All their drawings and paintings and photographs without a technological thing in them.

Actually, the defiance with respect to technical matters usually crystallizes in people's minds as mysticism. Pirsig explains it the following striking way:

> The "it" is a kind of force that gives rise to technology, something undefined, but inhuman, mechanical, lifeless, a blind monster, a death force. Something hideous they are running from but know they can never escape.

Only "peculiar" people accept to deal with "the beast", or are even fascinated by it. Or maybe it's the bunch of others which is "peculiar", by believing there is a beast in lifeless objects? Maybe Pirsig's first reaction was the correct one?

Among the counter culture mob, a sub culture existed within the hippy community, and the concept of personal computer, for instance, owes a lot to the technophile hippies, who wanted to be independent from the established mainframes (read IBM!). I have personally known a hippy, in the seventies, who used to return to teaching Fortran for a few days, when money was missing to run his house.

Now, of course, the hippy movement was a very strong one, which deeply and durably impacted our world, even if most of its dreams were naive, and irrealistic. Today, it sounds a little bit odd to think that software evolutions are deeply influenced by the evolution of thought. Well, it may sound odd, but it is still true, and these influences should be considered with much care, as software development is a very delicate matter, in which there is not a lot of space for approximation.

Our society has evolved since the early '70s. In the early '70s, the goal was to offer access to affordable goods to a majority of people. Quality was not the number one preoccupation. Now people are more preoccupied by their health, their weight. Organic food, pure products are what the market expects. "Pure" and "Thin" are the words that sell, and their momentum are very powerful. "Rich", "creamy" are negative, while "light", "pure", "no additives" are considered valuable properties. Consider haute couture top models, they do not look like Rubens or Renoir feminine ideals, anorexic is an adjective that applies better to them than healthy. Less and simple is now better. So it is, in the software business. There even used to be a company, sell-

ing software quality assurance tools called "Pure Software", now part of IBM, with products such as "Purify" and "PureLink".

Let us take modern examples to illustrate non technical influence on technology: XML, Java and Graphical User Interfaces (GUI) development.

2.8.1 XML, and the quest for purity

In the beginning God created the heavens and the earth. In the recent history of software, the origin of data formatting is SGML. SGML stands for "Standard Generalized Markup Language". It was invented by Charles F. Goldfarb in 1974, following "GML", with the goal to express a document in a structured manner. SGML provides a notation, which allows to manipulate a document as a composition of recursive sub-structures. After all, we all know that a book is divided in chapters, sections, subsections, paragraphs etc. This structure is conceptually independent from the way it is presented to the reader. SGML intended to allow this separation, splitting away structure and presentation. Also, it proposed a way to describe *a priori* the structure of a document, as a kind of grammar, under the form of a DTD (for Document Type Definition). A given text, expressed using SGML could then be validated against the DTD to check if it obeyed the generic structure of the class of documents it was supposed to belong to.

Goldfarb wrote in 1971:

> The principle of separating document description from application function makes it possible to describe the attributes common to all documents of the same type. ... [The] availability of such 'type descriptions' could add new function to the text processing system. Programs could supply markup for an incomplete document, or interactively prompt a user in the entry of a document by displaying the markup. A generalized markup language then, would permit full information about a document to be preserved,

> regardless of the way the document is used or represented.

SGML is an International Standard (ISO 8879) language. Its most famous sequel is running on most of the workstations in the world, it is HTML, and it is the base for most of the pages stored on Internet. HTML derived notably from SGML. It uses the same markup principles, but mixes a great deal of presentation information. At the time HTML was invented, the problem was to roll out Internet, and many concrete concerns existed. The initial wish to have the structure, and only the structure, of a document described in HTML, was negligible.

Needs have evolved, and the requirement to structure information has extended from documents to messages exchanged by computers on local area network, but also across the planet. SGML has been considered as a good base to build this. One of its other sequels is XML, and its purpose is to allow these exchanges.

XML is a "Pure SGML". It is 100% pure application data. It deviated from HTML, by firming up the structuring nature of SGML, and prohibiting any cosmetics which could be mixed up with the information.

Excerpt from a comment on the XMLEdge 2001 conference:

> Goldfarb's vision for XML is pure and based on freedom, not proprietary solutions.

There is a book called "Pure XML", by Aaron P. Rorstrom and George Doss [45]. This book belongs to a entire series, published by Sams, which contains also the following: "Pure C#", "Pure Java 2", "Pure C Programming", "Pure C++ Programming", "Pure Visual Basic", "Pure JavaScript", etc.

Anthony Channing, in August 2000, stated on the xml-dev mailing list:

> XML will become devalued and meaningless (although not useless) unless everyone sticks to pure XML. If

> the goal of XML is to improve the intercommunication of products within the computing world then it will only work if the rules are followed.

Quotes are numerous, this document would not be sufficient. "Pure XML" reaches on google 20,800 hits, while "pure HTML" (a contradiction in itself, as we saw) reaches 98,000 hits, and "pure SGML" reaches a mere... 290! The ratio of "pure SGML" with respect to "SGML" alone and of "pure XML" with respect to "XML" alone are in the 1 to 5 order of magnitude. Clearly "purity" was not a selling concept at the time of SGML...

This is in perfect philosophical synch with the reasons driving us to do our shopping at the supermarket... Pure food did not sell in the seventies, it had to be "rich"... How strange.

2.8.2 Java

Java is a programming language which, during the recents years, benefitted most from the "thin" momentum. The comments below do not constitute a criticism of Java, but rather intend to show how the most improbable technology, very low level technology, such as a programming language, can be influenced by society, and the evolution of mentalities.

When Java started to be promoted by Sun Microsystems, it was sold as a *light* technology to produce *thin* graphical user interfaces. It was presented as a reaction to former programming languages, like C++. C++ is a technology for professionals, who want to control very precisely what they do, while benefitting from modern software engineering practices such as object-oriented programming. It is, like its ancestor, C, used by a variety of respectable professional software corporations for building very sound solutions. The Java compiler, and the engine which executes it, the Java virtual machine, are, by the way, both written in C, not in Java. Oracle, the reputed relational database editor, push their customers very hard to use Java, and claim its universality, and efficiency. I personally experienced asking top management at Oracle, in California, when they would use Java as the technology for the core of *their own* flagship relational

database product. It was a nasty question, and the reaction of my interlocutor was no surprise. The person opened her large eyes wide full of incredulity, for such a peculiar question. Of course, the answer was a loud and clear: *never*. Oracle, for their own needs, for the core of their technology, prefer the "rich and creamy" C and C++...

You have to realize that when you use a programming language, nobody forces you to use everything. You can use any kind of subset you want. It is just like English or French, different people use a different subset. Few, if any, master the entirety of vocabulary and grammar. Having more constructs is not like carrying a heavier rucksack. You can simply ignore what you do not understand or what you do not master.

It is exactly the same in the world of programming languages. Nobody is forced to use all the constructs a programming language proposes. Even within the C++ community of developers, many of them (all?) use a subset. How then can "less" be "better"? By definition, it can only be worse.

Trying to prohibit people from using a given construct, because you do not use it yourself, is highly worrying. Doesn't it come close to fanatism? Well, this fanatism is pretty common in the peculiar world of software development. Every software developer has their own stories in memory. Some corporations prohibit double pointers in C, and allow only simple ones, because double pointers are "too complicated"... Some companies prohibit multiple inheritance in C++, because it is supposedly a very silly construct. Actually the people doing this clearly never spent time to understand what it is, and the subtleties around it. To some extent, the message "less is better" pleases people who have a hard time understanding too abstract matters. It goes on the same direction as anti-intellectualism. The bad news is that software development is a purely abstract activity, and the people who are repelled by it are already surrounded by abstraction.

As a matter of fact different technologies are meant for different people and different purposes. Java "surfed" so much on the "light" wave, explaining that simpler was better and that other

programming languages did not deserve so much complexity, that some level of fanatism arose within the Java community. As an example, Java, up to its 1.4 version, did not contain any means to describe genericity. Genericity is one of the interesting, but very difficult tools that were invented for modern software engineering. The simple fact that Java did not embed any way to specify generic constructs, was considered by its users as an asset. Less is better. Java did not *require* this, as its model did not require it. I've heard the most brilliant people explain that no kind of genericity was necessary in Java. Genericity (in a lighter form than the one proposed in C++...) appeared in Java v1.5, much to the puzzlement of those who predicted it would never be the case, and for whom Java was already perfect.

Java is an interesting example, also because it surfed on the "pure" word as well. Java evangelists — interesting term in itself — have succeeded in imposing the expression "100% pure Java", which borrows directly from its coffeinated homonym. "Better" started meaning "with more Java", and if possible, only with Java. And people were, and are still finding rational technical explanations for this. They usually forget to realize the obvious fact that Java relies on an operating system, Windows or Unix for instance, which is not, itself, written in Java, so 100% Java is meaningless. Interestingly enough, a very successful graphical toolkit, SWT, is attracting more and more attention from developers, because it is clearly faster than its predecessor, Swing. Swing is supposed to be 100% Java, while SWT contains only a "thin" layer of Java typically on top of C or C++ code...

How come the obvious is not obvious to the most intellectual people?

Another claim of Java evangelists, which took really well among Java users, is the absence of "pointer" in Java. Pointers are as old as information technology, as old as Alan Turing's works. Most, if not all Java developers are propagating the same motto "Java does not have any pointers". How come Java allows to set a reference to an object to the `null` value, accepts the comparison of a reference to `null`, and has an exception called "NullPointerException", that the same developers have

seen numerous times? Null *Pointer* Exception!!! At the time of writing, google returns 8,760 hits on the search of *java "no pointers"*, and 4,980 for *java "no pointer"*... 243,000 pages refer to "NullPointerException"...

The thing is that Java, semantically, not only *has* pointers, but it has *only* pointers. It is lacking a reference mechanism which guarantees that the reference is never null. But it indeed has no pointer arithmetics, and the superficial syntactic notation is not the usual pointer notation.

The striking ability of modern marketing to use the same techniques as mental manipulation, and build on the aspirations and trends of a society has created such a situation where the smartest minds are easily driven to the most obvious errors. Software developers are not isolated from the evolutions of the world, and are influenced by fashion too.

2.8.3 User interface development

During the last decade, the "thin" adjective, has also been a key selling point for graphical user interfaces technologies (GUI). Old practices were "heavy" while new ones, particularly those capable of running within a Web browser, were considered "thin".

In the world of graphical user interfaces, the yin has also its yang. "Thin" can also be seen as "poor" and "heavy" can also be seen as "rich". Of course, "heavy" does not sell, but often, it is what is required. Not always, of course, but often though. Let us clarify.

With the advent of Internet, a very popular way to interact with information has spread down to everybody's houses: the web browser. Naturally, it has been interpreted as the universal mean to access information. Suddenly the web browser became the universal tool not only to access Internet, but also to produce *all* graphical user interfaces. Of course, there would be a few negligible pieces of software like a text editor, or a spreadsheet, which would have real specific requirements, which would make a web browser inadequate for them. But all the rest would rapidly be replaced by web browsers. Every GUI in the world would

become "thin", and the world would be immediately better.

Well, let us come back to the root of things. Let us look at the "details". If we project ourselves back to before the advent of the web browser, and if we try to list the requirements the web browser would have to obey, it would be something like:

The users of the system are infrequent users, who are surfing on Internet, and switch from site to site, without usually spending much time on one of them. In cases when a user comes back regularly, the site would change at a high pace, and it would be just as if the user would switch from one site to another one. A given user would therefore not like to spend much time *learning* the system, and would prefer a stereotyped access to information, which would favor the absence of learning curve over ergonomy. These two are contradictory, because a better ergonomy involves more sophisticated interactions with the software, and interaction semantics require learning these semantics.

In a nutshell, a web browser is meant for an infrequent user, who does not want to learn anything beyond what he can learn by using the web browser during a few minutes.

A nice consequence of this is that a web browser can display pages which have been built without really using a programming language. This is thin thin thin.

All this translates into a very stereotyped ergonomy, which differs greatly from classical user interfaces. It also means that very intense users, who spend a meaningful part of their working day interacting with the GUI, are unlikely to be efficient, and are likely to be quickly upset by the tool.

In spite of the restrictive statement on the scope of use of a web browser, a number of very legitimate uses outside of Internet scope have arisen. As a matter of fact, the word "Internet" has been intentionally removed from the specification above, because the web browser will still be a very serious candidate as a GUI solution for infrequent users. "Thin" is well adapted to users who can cope with a "poor" experience.

The passion for universal solutions in IT is such that very few people identified the true constraints which gave birth to the web browser that we all know now. The frenzy around exploiting

web browser even for intense use has been truly incredible. What was the result?

The fact is that users are always right. No matter what you try to impose upon them, if they have requirements, they will obtain solutions to their problems, or they will move away. Intense users accept to spend some time in training, if this means getting a better productivity or a more comfortable work environment. Therefore, in the "intense" world of users, ergonomy prevails over reduced training. A "poor" experience is simply unacceptable, and a "rich" environment is mandatory.

What happened? Rich features had to be built on top of web browsers. To some extent, this has been true even within Internet world. Some web sites do not change so frequently, and to get fancier web sites, some solutions have had to be found in order to *compensate* the deficiencies of the "poor"/"thin" interfacing method a web browser proposes.

Embedding rich features within a web browser is difficult, and raises software engineering, network efficiency, and security problems. The vast majority of the solutions put in place involve using a programming language: JavaScript. The gradual implementation of more and more sophisticated interactions has often led to situations where enormous amounts of JavaScript code were developed. Unfortunately, JavaScript has hardly benefitted from any recent progress in software engineering, and hoards of developers are painfully experiencing software development with stone age tools.

At the end, gradually, the result is a rather poor interaction, with costly developments, and user interfaces which end up being extremely fat and slow. The guy who was supposed to be thin is now obese. This shows the risk of becoming a victim of fashion...

Some of the software tools we are using every day exist in two versions, light and rich, and nobody is surprised to have access to both. Among them are mail tools. If you process your mail at a reasonable pace, you certainly prefer to use your preferred Microsoft Outlook or IBM Lotus Notes solution. If you are on the road, or processing mail only sporadically, hotmail or google mail are, for instance, very interesting alternatives. If

somebody tried to force you to process your office mail using hotmail or google for hours a day, you would certainly become crazy... Surprisingly enough, that is exactly what lots of people have done in their own sphere of activity. This shows the weaknesses that anybody can have, even intellectuals, when they are under social and in particular, fashion, pressure.

More generally, marketing in technology borrows many of its techniques from sects, and modern technology marketers are even called "evangelists". The etymology of "evangelist" is biblical, and is a clear reference to religion. To sell technology better, it needs to be turned into a religion. If you attended some of the recent technology conferences, you may have noticed the very striking use of words like "community", "together", and repeated instructions like "stand up", "sit down", which are classical techniques to instrumentalize people. Similarly, marketing of technology is turning more and more towards *populism*, and selling mottos that the mass expect. Unfortunately, as explained in the former chapter, the successful subset of the IT population is only a very limited part of it, and populism can only be detrimental to people who become victim of it, as it does not show them the difficult path, while directing them towards the sliding way down to the abyss.

We must always have in mind Douglas Adams' joke:

> The Encyclopaedia Galactica defines a robot as a mechanical apparatus designed to do the work of a man. The Marketing Division of the Sirius Cybernetics Corporation defines a robot as 'Your plastic pal who's fun to be with'.

2.9 Elite as a bunch of marginals

For those who have meaningfully been exposed to developers, comparing some of the very best ones to marginals would seem very accurate. Being part of an "elite" and being a "marginal", if you take a neutral acceptation of "marginal", can sometimes be considered as synonym. Really brilliant people are scarce,

and their group constitutes only a margin of the population. As a matter of fact, the negative side of the word "marginal" applies to many members of this elite as well. Why so?

Developing requires a great deal of concentration, remember it is mostly an intellectual activity. The level of concentration reached by very talented developers brings them to a state that could be qualified as "withdrawal". Individuals who can reach this state are often never able to completely leave it! Also, even if some people would be capable of doing so, the state of withdrawal required is so scary to them, that they would not be willing to enter it. Douglas Adams, in the radio series of *The Hitchhiker's Guide to the Galaxy*, the secondary phase, "fit he ninth & fit the tenth", volume 6, track 14 says:

> It is often said that a disproportionate obsession with purely academic or abstract matters indicates a retreat from the problems of real life. However, most of the people engaged in such matters say that this attitude is based on three things: ignorance, stupidity and... nothing else.

Indeed, many developers — but not all of them — live in some kind of *retreat from real life*. What can I give as an example of that? Well, for instance the usual level of body hygiene we are all used to, as a consensus of today's life. This level of hygiene is sometimes not shared by some developers, and they can reach a state when they are so "withdrawn" that a pungent odor emanates from them. There is a famous story, from the old times, on Richard Greenblatt. Levy, in [33] recalls:

> Some hackers recall that one of the things Greenblatt's hacking precluded was regular bathing, and the result was a powerful odor. The joke around the AI lab was that there was a new scientific olfactory measure called a milliblatt. One or two milliblatts was extremely powerful, and one full blatt was just about inconceivable. To decrease the milliblatts, the story goes, hackers maneuvered Greenblatt to a place

> in the hallway of Building 20 where there was an emergency shower for cases of accidental exposure to chemicals, and let it rip.

As we already said, withdrawal from reality is not always a choice, not always a consequence of this very peculiar activity which is software development. Often, the natural wish to withdraw from reality is the powerful incentive to move to the development world. Withdrawal is not the consequence, but the cause. Levy, on this subject, says:

> [...] for one thing, many of the hackers were loners to begin with, socially uncomfortable. It was the predictability and controllability of a computer system — as opposed to the hopelessly random problems in a human relationship — which made hacking particularly attractive.

The wish for withdrawal is very close to mental illness. The section on Richard Stallman gives ample evidence on that. The ability to oscillate between madness and genius is a key capability to become a very bright developer. Douglas Adams, in H2G2 has a word on this:

> The border between madness and genius is very narrow.

2.10 Silence and intellectual work

In the introduction to this document, the case of the variations of productivity related to office space layout was reported. Today, there are still many corporations where it has not been understood that software development is an intellectual activity, and where it is somehow considered as manufacturing. As a consequence, they organize software production through Adam Smith's division of labor principle, and design their offices as if software production was something like mass production of textile goods. It is extremely common to see that offices are open

space, and people are operating their sewing machine — sorry, computer — to perform their daily duty.

If you admit that software development is intellectual, you have to conclude that, being intellectual, it requires an environment which is favorable to concentration. Tom DeMarco and Timothy Lister in "Peopleware — Productive Projects and Teams" [13] make an extensive analysis of what psychologists call the state of *flow*, which is precisely the concentration we are talking about. This state can be reached after a period of 15 minutes of quietness. Any disruption, even small, destroys 15 minutes to return to the state of flow.

McConnell [35] calls flow-friendly offices "Thinking-Oriented Office Space". He reminds that studies show that "productivity levels of developers who work in private, quiet, one or two person offices can be as much as 2.5 times as great as the productivity levels of developers who work on open work bays or cubicles". Among the scarce studies on the relationship between office layout and developers' productivity, we find the IBM Santa Teresa study [36]. This is what an architect exemplary work can be. It attempts to define what is the most adapted office layout for the specific task of software development.

Robert Pirsig, in ZAMM [40], follows the same track. He does not talk about software development, he takes motorcycle maintenance as an examplary case. And he wonders why some, if not most, mechanics are so bad. Eventually, he finds the reason:

> The radio was a clue. You can't really think hard about what you're doing and listen to the radio at the same time. Maybe they didn't see their job as having anything to do with hard thought, just wrench twiddling. If you can twiddle wrenches while listening to the radio that's more enjoyable.

And read this:

> An untrained observer will see only physical labor and often get the idea that physical labor is mainly

> what the mechanic does. Actually the physical labor is the smallest and easiest part of what the mechanic does. By far the greatest part of his work is careful observation and precise thinking. That is why mechanics sometimes seem so taciturn and withdrawn when performing tests. They don't like it when you talk to them because they are concentrating on mental images, hierarchies, and not really looking at you or the physical motorcycle at all. They are using the experiment as part of a program to expand their hierarchy of knowledge of the faulty motorcycle and compare it to the correct hierarchy in their mind. They are looking at underlying form.

This means that software developers who constantly listen to music while performing their work are not really, genuinely, completely, dedicated to it, and just like for motorcycle maintenance, will only be capable of performing lower grade work.

Development requires a sustained level of concentration, not only during minutes, but during hours. McConnell [35] compares developers being constantly interrupted to the situation of Albert Einstein receiving frequent calls from his manager telling him "Albert, we need that theory of relativity *now*! Hurry up!". Beyond the joke, we can find lots of references to the relationship between peaceful environments, concentration and intellectual work or activities:

Regularly, Bill Gates, with a bunch of other Microsoft employees, leaves the office, and makes "retreats", simply because this is favorable to more accurate reflection.

The good Edgar Allan Poe [41], in "The Purloined Letter" [41], goes even further, and recommends not only shutting down noise, but also light, and makes his hero, the marvelous puzzle resolver Dupin say:

> "If it is any point requiring reflection," observed Dupin, as he forebore to enkindle the wick. "we shall examine it to better purpose in the dark."

While this is probably a little bit radical — Poe was pretty extreme —, it means that the brain's attention should be focused on its duty, and not be entertained at the same time, by other stimuli.

Ralph Waldo Emerson in "The American Scholar" (1837) [16] opposes to the result of parcellized labor inherited from Adam Smith:

> In silence, in steadiness, in severe abstraction, let him hold by himself

Very serious scientific studies have been conducted on the relationship between noise and inability to learn. For instance, the BBC reported on 2^{nd} of June 2005:

> A team from Barts and the London NHS Trust looked at data on more than 2,800 children living near Heathrow and other airports in Spain and the Netherlands.
>
> The Lancet study found each five decibel increase in noise level was linked to children being up to two months behind in their reading age.
>
> A US expert said the study supported previous research findings.
>
> The children, all aged nine or 10, attended schools near to London's Heathrow Airport, Schiphol in the Netherlands and Barajas in Spain.
>
> [...]
>
> Reading age was delayed by up to two months per five decibel increase in noise levels in the UK children studied, who attended schools in the boroughs of Hounslow, Hillingdon and Slough, and up to one month in the Dutch children.
>
> [...]
>
> This translates to a delay of up to eight months in a child's expected reading age.

> [. . .]
>
> [Professor Stephen Stansfeld] highlighted one study which looked at children living near to the old Munich airport in Germany, before and after it was closed down.
>
> "Children attending schools near the airport improved their reading scores and cognitive memory performance as the airport shut down, while children going to school near the new airport experienced a decline in testing scores."

The inability to concentrate seems related to distractions, which disrupt the thinking process. This is shown by another study, carried out by TNS Research and commissioned by Hewlett Packard. CNN reports:

> The survey of 1,100 Britons showed:
>
> - Almost two out three people check their electronic messages out of office hours and when on holiday
> - Half of all workers respond to an e-mail within 60 minutes of receiving one
> - One in five will break off from a business or social engagement to respond to a message.
> - Nine out of 10 people thought colleagues who answered messages during face-to-face meetings were rude, while three out of 10 believed it was not only acceptable, but a sign of diligence and efficiency.
>
> But the mental impact of trying to balance a steady inflow of messages with getting on with normal work took its toll, the UK's Press Association reported.
>
> In 80 clinical trials, Dr. Glenn Wilson, a psychiatrist at King's College London University, monitored the IQ of workers throughout the day.

> He found the IQ of those who tried to juggle messages and work fell by 10 points — the equivalent to missing a whole night's sleep and more than double the 4-point fall seen after smoking marijuana.

This is probably why Donald Knuth never answers e-mail in real time anymore, and responds through snail mail. In an interview given to Computer Literacy Bookshop Inc., on the 7^{th} of December 1993, he says:

> I spent fifteen years using electronic mail on the ARPANET and the Internet. Then, in January 1990, I stopped, because it was taking up too much of my time to sift through garbage. I don't have an email address. People trying to write me unsolicited email messages get a polite note saying "Professor Knuth has discontinued reading electronic mail; you can write to him at such and such an address."
>
> It's impossible to shut email off! You send a message to somebody, and they send it back saying "Thank you", and you say "OK, thanks for thanking me..."
>
> Email is wonderful for some people, absolutely necessary for their job, and they can do their work better. I like to say that for people whose role is to be on top of things, electronic mail is great. But my role is to be on the bottom of things. I look at ideas and think about them carefully and try to write them up... I move slowly through things that people have done and try to organize the material. But I don't know what is happening this month.

E-mail is not the only culprit, telephones have been finger-pointed too because they prevent being on the *bottom* of things.

We could cover pages with references to silence and brain operation. Meditation, for instance, is an example of brain operation that is usually associated with silence. This is true for many religions, which have often pushed for vows of silence.

Let us finish with Gandhi's words:

> It has often occurred to me that a seeker after truth has to be silent. I know the wonderful efficacy of silence.

and:

> The virtues and qualities of silence are often lost in the culture of ceaseless noise, loud advertising and our neighbours and friends speaking senselessly to tranquilise their own minds.

2.11 Conclusion

At the end of this chapter, it is important to go beyond the funny side of the biographies and stories which were presented. The goal of this chapter was not to let the reader believe that developers are freaks, and that they are affected by all possible mental diseases. To say the truth, I must admit that a few of them indeed are freaks or mentally sick, or both of them, but the vast majority are perfectly sane, and also capable of combining very strong concentration and creative abilities. The word "genius" is often quite deserved. Diving into their daily job, into the abstraction they are used to, is unpleasant to many people, if not dangerous. This is actually very comparable to deep sea diving. Pirsig is talking about his former personality, that was cured through electroshocks, Phaedrus. Being capable of diving without becoming Phaedrus, is an admirable achievement. Being capable of keeping the holistic view of problems, which is required by the most efficient work organization—project division—, and taking care of the implacable utterly fine details computing requires stretches one's brain to a point very few people are capable of reaching.

Making use of genius is what software production requires. This cannot be done naively.

Bibliography

[1] Ivan Aaen, Peter Bøtcher, and Lars Mathiassen. The software factory: Contributions and illusions. pages 407,433, Oslo, 1997. Proceedings of the Twentieth Information Systems Research Seminar in Scandinavia.

[2] A. J. Albrecht. Measuring application development productivity. page 83, Monterey, CA, Oct 14-17 1979. IBM Applications Development Symposium, GUIDE Int and Share Inc.

[3] Charles Babbage Esq. A. M. *On the Economy of Machinery and Manufactures*. Carey & Lea, 1832.

[4] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[5] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[6] Stewart Brand. We owe it all to the hippies. Forget antiwar protests, Woodstock, even long hair. The real legacy of the sixties generation is the computer revolution. *Time Magazine Domestic*, **145**(12), Spring 1995.

[7] Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[8] Frederick P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. pages 1069–1076, Dublin, 1986. IFIP.

[9] Thomas Capers Jones. *Assessment and Control of Software Risks*. Yourdon Press, 1994.

[10] Alan Cooper and Robert Reimann. *About Face 2.0, The Essentials of Interaction Design*. Wiley Publishing, Inc., 2003.

[11] James O. Coplien. *Advanced C++ programming styles and idioms*. Addison-Wesley, 1992.

[12] Michael A. Cusumano. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, And Manages People*. Free Press, 1998.

[13] Tom DeMarco and Timothy Lister. *Peopleware — Productive Projects and Teams*. Dorset House Publishing Co., 1999.

[14] Peter J. Denning and Robert Dunham. The core of the third-wave professional. *Communications of the ACM*, **44**(11):21–25, November 2001.

[15] Edsger Wybe Dijkstra. *The Humble Programmer*. ACM Turing Lecture, 1972.

[16] Ralph Waldo Emerson. The American Scholar. An Oration delivered before the Phi Beta Kappa Society, at Harvard, August 31^{st}, 1837.

[17] René-Antoine Ferchault de Réaumur. *Mémoires pour servir à l'Histoire des Insectes*. Imprimerie Royale, Paris, 1734.

[18] Gordon E. Forward, Dennis E. Beach, David A. Gray, and James Campbell Quick. Mentofacturing: a vision for American industrial excellence. *Academy of Management Executive*, **5**(3):32, 1991.

[19] Robert L. Glass. *Software SOLILOQUIES*. Computing Trends, 1980.

[20] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002.

[21] Robert L. Glass. Negative productivity and what to do about it. *IEEE Software*, pages 95–96, September/October 2008.

[22] Michael Hammer. Reengineering work: Don't automate, obliterate. *Harvard Business Review*, pages 104–112, July/August 1990.

[23] Michael Hammer and James A. Champy. *Reengineering the Corporation: A Manifesto for Business Revolution*. Harper Business Books, 1993.

[24] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1999.

[25] Richard Hofstadter. *Anti-Intellectualism in American Life*. Vintage Books USA, 1966.

[26] Ludvig de Holberg. *Nicolai Klimii Iter Subterraneum Novam Telluris Theoriam Ac Historiam Quintae Monarchiae Adhuc Nobis Incognitae Exhibens E Bibliotheca B. Abelini*. J. Preuss, Copenhagen & Leipzig, 1741.

[27] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[28] Jack Kérouac. *On the road*. Penguin Books, 1991.

[29] Donald Ervin Knuth. *The Art of Computer Programming*. Addison-Wesley, 1981.

[30] Donald Ervin Knuth. *3:16 Bible Texts Illuminated*. A-R Editions, 1991.

[31] Donald Ervin Knuth. *Computers & Typesetting, Volumes A-E Boxed Set*. Addison-Wesley Professional, 2000.

[32] Bill Landreth. *Out of the Inner Circle: A Hacker's Guide to Computer Security*. Microsoft Press, 1985.

[33] Steven Levy. *Hackers: Heroes of the Computer Revolution*. Anchor Press/Doubleday, 1984.

[34] Karl Marx. *Capital: A Critique of Political Economy*. The Humboldt Publishing Company, 1886.

[35] Steve C. McConnell. *Software Project Survival Guide*. Microsoft Press, 1998.

[36] Gerald M. McCue. IBM's Santa Teresa Laboratory—Architectural design for program development. *IBM Systems Journal*, **17**(1), 1978.

[37] Harlan D. Mills. *Software Productivity*. Dorset House Publishing Co., 1988.

[38] Kevin D. Mitnick. *The Art of Deception: Controlling the Human Element of Security*. Wiley, 2002.

[39] Jean-Louis Peaucelle. *Adam Smith et la division du travail : La naissance d'une idée fausse*. L'Harmattan, 2007.

[40] Robert M. Pirsig. *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*. William Morrow & Co, first edition, 1974.

[41] Edgar Allan Poe. *The Works of Edgar Allan Poe in four volumes*, **Vol. I.** W. J. Widdleton, 1864.

[42] William Poundstone. *How Would You Move Mount Fuji? Microsoft's Cult of the Puzzle*. Little, Brown and Company, 2003.

[43] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program. Technical report, Fakultät für Informatik, Universität Karlsruhe, March 2000.

[44] R. Réaumur and A. de Ferchault. *Art de l'Epinglier avec des additions de M. Duhamel du Monceau et des remarques extraites des mémoires de M. Perronet, inspecteur*

général des Ponts et Chaussées. Paris, Saillant et Nyon, 1761.

[45] Aaron P. Rorstrom and George Doss. *Pure XML*. Sams, 2001.

[46] Murray Rothbard. *An Austrian Perspective on the History of Economic Thought*. Ludwig Von Mises Institute, 2006.

[47] Oliver Sacks. *The man who mistook his wife for a hat and other clinical tales*. Touchstone, 1998.

[48] Alfred P. Sloan. *My years with General Motors*. Doubleday & Co, 1964.

[49] Adam Smith L. L. D. *An Inquiry into the Nature and Causes of the Wealth of Nations*. W. Strahan and T. Cadell, London, 1776.

[50] Alfred Spector and David Gifford. A computer science perspective of bridge design. *Communications of the ACM*, **29**(4):268–283, 1986.

[51] Randall E. Stross. *The Microsoft Way: The Real Story of How the Company Outsmarts Its Competition*. Perseus Books Group, 1997.

[52] Taligent. *Taligent's Guide to Designing Programs, Well-Mannered Object-Oriented Design in C++*. Addison-Wesley Publishing Company, 1994.

[53] Frederick Winslow Taylor. Shop management. *Transactions of the American Society of Mechanical Engineers*, **24**:1337–1480, 1903.

[54] Frederick Winslow Taylor. *The Principles of Scientific Management*. Harper & Brothers Publishers, 1911.

[55] David Thielen. Writing exceptional software. *Software Development International*, pages 57–59, Winter 1991.

[56] David Thielen. *The 12 simple secrets of Microsoft management*. McGraw-Hill, 1999.

[57] Alvin Toffler. *Future Shock*. Bantam Books, 1970.

[58] Alvin Toffler. *The Third Wave*. Bantam Books, 1990.

[59] William H. Waddell and William Bodek. *Rebirth of American Industry - A Study of Lean Management*. PCS Press, 2005.

[60] Sam Williams. *Free as in Freedom: Richard Stallman's Crusade for Free Software*. O'Reilly, 2002.

Index

ISBN 978-0-9798-3404-2