Erik Piñeiro

The Aesthetics of Code

On excellence in instrumental action

Fields of Flow

for Elvira

Erik Piñeiro

The Aesthetics of Code

On excellence in instrumental action

The fisherman who intends to fish for salmon needs to consider many different factors that will influence his multiple decisions. He needs to cast the fly before the salmon in such a way that the fish cannot resist the temptation and takes it, preferably taking the whole lure into its mouth. It is here that the fisherman's technique makes all the difference and needs to be varied according to prevailing conditions. Sometimes it should be enticing, sometimes brusque, perhaps hesitant and possibly provocative. The range for personal expression is vast. The angler has the choice of different lines, floating or sinking. He can skim the water's surface with the traditional feather, hair wing or tube flies. The fisherman can cast across the run or with the current, and he can retrieve quickly or slowly. He can have the fly land directly in front of the salmon, have it sink down to the fish or pull it up to it. Many other options are open to the angler, and the range of choice in fishing techniques is truly unbelievable.

Hitch Craft, 1994
North Atlantic Salmon Fund Reykjavik, Iceland

ACKNOWLEDGEMENTS

Finding yourself at the point where you are allowed to defend your thesis is all very well, but I must admit that the more I think about it, the less important it feels. Holding a ready-to-publish manuscript in your hands is nice, but it is hardly comparable to recalling all the things that have happened on the way there. The five and a half years of doctoral studies have been a wonderful experience, and I am happy there is a place where I can express my gratitude to those who have made them so. If it were up to me, I would bore you with a chapter on each person, but I'll be brief.

Five and a half years ago, Professor Claes Gustafsson was explaining to me, as he has done before to numerous other students, how one's research idea changes throughout the PhD studies. How one enters the process with one question and exits it, a few years later, with a rather different one. This, of course, has happened to my research question, but more importantly, it has also happened to me, as a person. Thank you Claes, for offering me a place at INDEK, for making doctoral life so very enjoyable, for all the advice – academic and otherwise –, and for the car (even if, strictly speaking, it wasn't yours).

Just over thirty-three years ago, Bertil Guve and I were both floating upside-down inside our respective mothers while they went out together in Madrid. Then we were born and were friends, but a few years later he moved to Stockholm. Many years later, I moved to Stockholm and he was here to welcome me. Thank you Bertil. For your help when I arrived here, and for convincing me and Claes that this was a good idea. We have made many journeys together, both to the inside and to the outside, every one of them has been moving and enlightening.

The first time I saw Marcus Lindahl he was wearing snake skin cowboy boots. I can't quite recall what I thought right then, but it was definitely not that you would become such a great friend. There is a chance we would both have defended our thesis earlier had we not been room mates, but then I would have missed all the ideas, both the sharp and outlandish ones, you have offered through the years. Besides, I never felt I was in a hurry.

To engage in an academic argument with Alf Rehn is always a stimulating enterprise, at times even a daring one, but it is not quite as exciting as discussing life, the world and everything else. Thank you for your advice, for the books I've borrowed and for reading the successive manuscripts.

Around Claes there has been a group of PhD students, of which I have formed part. We have discussed articles, books and each other's work. These inspiring seminars have played an important role in the writing of this thesis. Claes, Alf, Bertil and Marcus were there, but they were by no means the only ones. Thank you Tina Karrbom, Anna Jerbrant, Fredrik Markgren and all those who have taken part in them. During the last year or so, the seminars have become even better. The new wave of PhD students has brought new friends, more opportunities for motivating discussions, and more fun. Thank you Helena Csarmann for the talisman, Sven Bergvall for your thousand helping hands, Charlotta Manckert for a few well-timed hand grenades, David Sköld for the vynil, Mikolaj Dymeck for the gaming, and Thomas Lennerfors for reminding me of the good things about Madrid. The best of luck to you all.

Now, old and new PhD students might well make INDEK interesting and fun, but were it not for Christina Carlsson, Caroline Pettersson, Jan-Erik Tibblin, Jessica Matz, Christer Lindholm and the rest of the administration team, there would be no department to speak of.

Thank you all, these years would not have been possible without your solid proficiency.

I have gained many insights from both reading and discussing with professors Sven-Erik Sjöstrand and Pierre Guillet de Monthoux, and with all the members of the research project FIELDS OF FLOW. My gratitude to them and to the The Bank of Sweden Tercentenary Foundation, whose generous financial support has made the whole project possible.

In the final stages of the thesis, Daniel Pargman read the manuscript very closely and held a very helpful seminar. Ali Qassim made the final adjustments to my English. Thank you both.

Without my parents, Sonja Högberg and Guillermo Piñeiro, none of this – nada de nada – would have ever happened. I can think of a thousand reasons to show you my gratitude but, for this time, I'll settle with thanking you for supporting me in all my projects, including this one.

Thank you Daniel and David Piñeiro, for all kinds of things, but mostly for those more than eighteen years of living together. Aunts, uncles and cousins, thank you for all the good times. Erik Olof Albert, morfar, thank you for the memories.

There are other fine individuals that have made life great during these years. Thank you: Gemma, Björn and Nina, for unplanned and chaotic dinners in warm company; Krim, for loooong walks and heated discussions; Maja, for taking us to the forests; Omar, Rebecca and Ruben Rashid, for the music, the insights, the hospitality, and for London; Ulla, for all kinds of help with the apartment; Javier, por el jamón y el acento; Lena, for welcoming us to your place; Johan, Teddy and Alex, for the kayaks; Isabella, Guillaume and Selène, for the food, the sea and the arguments; Alejandro, por el flamenco; Juan Diego and Rosi, for Nova Notio and the continued friendship; Lidia, for keeping the contact; Rosa, for your lessons, Monik, for so many things… one's life is made of other people.

It's true, life was fine before Ester. But being with you is a bliss, a great and undeserved gift. Ester, light of my life, you give me sunshine in a cloudy day, and the month of May when it's cold outside. Thank you for everything.

This book is dedicated to Elvira, who is not yet born, but who will, hopefully, be born before this book leaves the print shop. During the last weeks I have been much more concerned about you and your mother than about writing and, well, I think that the thesis has benefited from that.

Elvira was also the name of both my grandmothers, mormor y la abuelita. To them a remembrance.

Erik,
Stockholm, 23.09.03

I

Contents

coding. Coding is considered by some programmers as a peripheral concern but, on the other hand, it results in something tangible (code), allowing thus for good illustrations. Besides, however secondary it may considered by some programmers, we shall see that coding aspects are important enough to spark off heated disputes among them.

VI AESTHETIC IDEALS 159

Through an examination of coding and its results, this thesis has introduced the technically complex subject of beauty in programming but it is not my intention to go into a detailed explanation of the aesthetic ideals of programming since this would require too much technical overhead. However, this study would be incomplete without an overview of the most popular aesthetic attributes of software. I will present here the following ideals: cleanness, simplicity, tightness, consistency, structure and robustness. Programmers use these words (the adjectives) to describe the beauty of their preferred programs, or, perhaps more often, the qualities they pursue when writing software. As in the previous chapter, the purpose here is not to provide a comprehensive classification but to delve more deeply into the phenomenon by offering more evidence of the existence and significance of the private aspects of programming.

VII THE RELATIONSHIP BETWEEN INSTRUMENTAL AND INTRINSIC GOODNESS IN PROGRAMMING 201

The previous chapter about instrumental, semi-instrumental and intrinsic goodness provided a description of these three concepts with the aim of introducing the *notion of instrumental beliefs. This conceptual toolkit was hopefully useful when reading the two empirically intensive chapters that followed. In this chapter we return to the concepts of instrumental and intrinsic goodness (of code), this time using them to explore how programmers approach the relationship between the public and the private aspects of software.*

VIII INSTRUMENTAL BELIEFS 237

As advanced in the chapter about instrumental goodness, and as described in those that followed, programmers find themselves making decisions based on a mixture of aesthetic preferences, instrumental beliefs about 'what is best' (for the user, for the company and for their colleagues), and technical knowledge. This may result in behaviour that is difficult to explain unless we accept that programming is not an objective activity but one in which personal factors play an important role. This chapter considers the concept of instrumental beliefs through the study of one such behaviour: careful previous design. Is there a point in calling this rather widespread procedure a ritual?

IV COMMUNITY 259

The concept of 'programming community' can be found a little bit everywhere, including scientific articles in the IEEE Software Journal, where it is used to explain why some programmers hold strong opinions on technical subjects that have not been empirically solved. In this chapter we shall explore the relationship between what we have called 'private aspects', our gathering term for that kind of phenomena, and the existence of a *programming community. Bataille's concept of sacrifice as expenditure will be our link, the argument being that some of private phenomena of programming (e.g. writing beautiful software) can be interpreted as the manifestation of a constant and individual sacrifice that brings about a sense of community. The sacrifice may take different forms, but in all cases expresses the same: a concern for software itself (its intrinsic qualities), more specifically, an economically oblivious concern for software. Obliviousness, however, is not the same as opposition, sacrifices are not generally carried out in order to waste but in order to express something. In fact, they may make good economic investments, as some programmers insist.*

X PROGRAMMING AS SYMBOLIC ACTION 287

It is time to draw some conclusions, and in this chapter the argument is reviewed as an attempt at presenting programming as symbolic action. Writing a program is not only solving a computational problem (constructing a virtual machine that carries the intended function), it is also a process by which programmers create, reaffirm and communicate their world view and their place in it (through aesthetic preferences and instrumental beliefs, for instance); it is a way of expressing oneself. In this sense, choosing emacs instead of vi (see chapter 5) is not (only) the result of rational considerations but (also) a symbol of one's identity as a programmer. Writing software looks, from this perspective, more like practising a religion than like calculating.

Research in the management of software development, and research in programming in general, should not ignore the personal aspects of programming. Programming managers, in turn, should not deal with them as simply something to be suppressed. I hope this much is clear after reading the thesis; but, given the picture of programming we have witnessed, is there anything we can say about other instrumental activities? Notably, about management? And is there anything to be said about technology at large?

1
Introduction

Software is one of the essential elements of Western society, which has made it the subject of studies of many different kinds. A large number of scholars dedicate time to both the technological and the human aspects of software, including studies of the process of software creation. However, most of these studies seem to assume that programming is an objective endeavour, a sort of deductive process almost, and that code is a 'neutral' artefact (whose usefulness is its only significant characteristic), rather than a creation to admire and to be proud of. Accordingly, they overlook the consequences that influences of the personal aspects of software have on programming, and, hence, in the management of programming projects. This study concentrates on the programmers' personal relation to their own creations, and although its immediate purpose is to offer an account of the private aspects of programming, it contributes to the library of management of software development projects, and perhaps also to our attempts at understanding the nature of instrumental action at large.

The Economist opened its June 21st 2003 Technology Quarterly Report with an article on software development, more specifically on bug-detecting systems. According to this article, a researcher at the America's National Institute of Standards and Technology has estimated that "software bugs are so common that their cost to the American economy alone is $60 billion a year or about 0.6% of gross domestic product." The same Institute has also calculated that "80% of the software-development costs of a typical project are spent on identifying and fixing defects." Further, the article explained that programming bugs are better found and fixed as early in the development process as possible, since the costs of fixing them escalate as the project progresses. "Djenana Campara, chief technology officer of Klocwork, a young firm based in Ottawa, Canada," proposes the following rule of thumb: "a bug which costs $10 to fix on the programmer's desktop costs $100 to fix once it is incorporated into a complete program, and many thousands of dollars if it is identified only after the software has been deployed on the field. In some cases, the cost can be far higher: a bug in a piece of telecoms-routing equipment or an aircraft control system can cost millions to fix if equipment has to be taken out of service." (cited from the same article)

Needless to say, a number of people are working on techniques and systems that minimise the amount of bugs generated by programmers. The article in *The Economist* centres around the idea of using applications (programs[1]) that detect bugs as the code is written. These applications scan the code being written and apply known algorithms to detect errors. Not all kinds of bugs are detectable in this way, of course, but the proponents claim that these kind of applications can make a difference. "Opinions", *The Economist* continues, "are divided as to whether programmers will welcome or reject such tools" (ibid). And this leads to the subject of this thesis. This study is not going to examine bug-detection methods, neither is it going to propose models to minimise the number of bugs per line. It does, however, intend to examine how programming is experienced by programmers. In other words, this thesis does not deal with bugs but with the activity of programming, more specifically with the ways in which programmers relate to the code they write. The insights gained should help managers decide on the question of "whether programmers will welcome or reject such tools."

There is no possible way to know a priori how each individual programmer will react towards a monitoring system that tells her (and her boss, and her colleagues) how well she is doing. In fact, I don't think anyone is really interested about this... on the other hand, what everyone wants to know is, on the other hand, whether the implementation of these automatic systems will actually result in better (less buggy) software. My impression, and this will form the basis of my argument in the next hundred or so pages, is that those who firmly believe in the utility of those systems base their conviction on a too narrow vision of programming. They assume that programming is something strictly instrumental (i.e. done only in order to achieve a goal and with no value in itself), that the only important aspect of the whole programming effort is whether the application runs or not (and has as few bugs as possible). From this perspective, programmers should clearly welcome any tool that helps them reach more swiftly the final bug-free application. A commonsensical argument, really, but one which is based on the wrong assumption: programming is not solely, perhaps not even mostly, about creating programs that run; it is also about personal expression and belonging.

Programmers have a creator-creation relationship to their code, and whether that code fulfils someone else's expectations (users, managers, for instance) is only an aspect of that relationship. Their code reveals a number of things about them, such as their skills, their technical preferences, their beliefs about what can be done with software, in a word, about the kind of programmers they are. The implementation of a monitoring system that reports the number of bugs contained in their code is likely to change their programming routines but, in my opinion, it is impossible to know what the results of that change will be. Will they actually use it as a bug-detecting tool? And in that case, will they become over-secure and lazy and write programs with even worse bugs that the system does not detect? Or will they instead make a game of the whole thing, trying to beat the system, introducing bugs that it cannot detect (and exchange bravados about their superiority over the machine)?

On the other hand, this is the way technological innovation often moves: through the creation of artefacts of uncertain use and whose consequences cannot be specified a priori. So I would never suggest that it is wrong to develop bug monitoring systems, only that we must be careful with the assumptions we make when assessing their utility.

As the reader has probably perceived, the issue of bug-detection is, despite the magnitude of its economical consequences, only secondary here. The aim of this thesis is to explore how programmers relate to the programs they work with, and to show that this relationship includes a concern for both the instrumental and the intrinsic qualities of code. They do not only want their code to work, they also want it to speak well about its creators (themselves).

There are more ways to explain what this thesis is about. At a most immediate level, it is about human aspects in software development. At a deeper level, it could also be about something (apparently) completely different: the fertility of fantasy and its role in the processes of technological innovation. Or about how anarchic human invention can flourish in as arid environments as that of computing logic. Or, in Feyerabend's terms, it could be about the impossibility of curbing the abundance of life.

This is, hence, a thesis about programmers and their relationship to software. And when I say software, I do not mean 'the uses of programs', or 'the business of software', or 'the appearance of applications on the screen', but to the *code itself*. This is, as we shall see, a colourful relationship, rich in nuances, that gives raise to heated disputes, "holy wars", vanity and other passionate manifestations. The image or programming projected by all these phenomena should dissipate the notion that it is only about solving complex logical puzzles and that programs are flat creations. The reader will hopefully see that software is not the only result of objective calculations, but also of the very human desire to create and to express oneself.

In a way, this a thesis not only about but also *for* programmers, who might find it interesting to read, for a change, about *the experience of writing code*, instead of about how to complete your programming projects in time and within budget, or how to write truly useful applications, or how to sell software, or how software is going to change business models all over the world. However, the purpose of this book is not to teach programmers anything about programming. My work has been to analyse the experience of programming, not with the aim of suggesting methods to make it more efficient but with the aim of bringing to light its personal aspects. Hence, the lessons that this book may offer will not make anyone a better programmer, but they will help widen the picture of what programming is.

This is also a thesis with something to say to programming managers, and other people whose profession puts them in contact with (some might say 'at the mercy of') programmers. Not that they will find here any recommendations as to how to carry out their jobs, no last-page bullet-summary with ten advices for their negotiations with programmers; instead they will find a description and an analysis of the experience of programming. Programmers sometimes complain about managers 'not getting it'. What they mean is that 'managers' (in a very general sense) are failing to understand the private aspects of programming. If managing is the art of understanding the relationship between the goals, the resources, and the circumstances that surround them (and not the application of rules and formulae) then this thesis is the place where to look for insights about the management of programmers: a book where the programming experience is explained by those who do it.

From the scholarly perspective, this thesis carries the programmers' voice, something that, surprisingly, has not yet been carried out with the care and attention it deserves, even if there has been some interest in ethnographic studies – see, for instance, (Cook, et al. 1993), or the more recent (Association for Computing Machinery 1997). Furthermore it suggests a conceptual toolkit with which to analyse the programming experience. This toolkit is essentially based on the notions of religion and beliefs (the central reference being Geertz), and on Bataille's work on sacrifice and community. You could say that what I do is propose a metaphor: that of programming as a religious practice. I shall elaborate on this later.

As for the programmers' voice, it should not only be interesting to those scholars in the field of technology management but also to those interested in technology at large. I am thinking more specifically about philosophers of technology, or anyone that reflects on what technology is and on what it both does and says about human nature. Is technology something intrinsically reductive (human nature is being slowly cornered)? Is it a sure sign of the supremacy of rational (see objective and dispassionate) thought? This thesis does not answer these questions, but it does deal with their most elementary subject (technology), and from a (strangely so) unusual perspective: that of the passionate engineer who loves to create, not for the good her creations may do to the world but for the sheer pleasure of creating.

But let me first entertain you with a few remarks about software and its role in our society. Is the voice of programmers, and their personal experience of programming, really so important?

SOFTWARE IS EVERYWHERE

Software is everywhere. It may be more or less obviously present, but it is practically everywhere. Wherever there is a microprocessor, there is software. And believe me, there are microprocessors all around you, not only in computers but also in cars, mobile phones, cameras, microwave ovens, minidisk players and all other kinds of objects. Some important actors of our society rely nowadays almost totally on the correct functioning of software systems: hospitals, air-traffic centres, trains, telephone switches, tax-collecting institutions… few manufacturing processes exist today in which software does not play an essential role, running anything from small robot-arms to comprehensive systems for production control and planning. Paying with your credit card, or withdrawing money from a cash dispenser requires the existence of a vast software system; although perhaps not quite as vast as the one that makes it possible to call your family in Stockholm from the taxi driving you to the Bangkok International Airport.

All those systems have to be programmed and maintained, which is the task, so to speak, of the software industry. The enormity of this task is hard to over estimate, at any given moment there are hundreds of thousands of engineers, technicians, managers, clerks, students and directors of boards working to make sure that the software systems upon which our society depends on, run. In fact, the real significance of programming, and of this thesis, does not stem from the economic magnitude of the software industry but from the ubiquity of programs. This omnipresence makes of software one of the fundamental elements of our society, in fact, society as we know it at the beginning of the 21st century could not exist without software. The pervasiveness of microprocessors and software has modified a great number of routines and processes, and when people say that we now live in a 'post-industrial' era, or that we now must follow the rules of a 'network economy', I tend to think that we could also say, even if it sounds more awkward, that we live in a software-enabled-era. Not all progress has been for the good, weapons of massive destruction are impossible to create and deploy without some sort of computing power, but I am not really interested in a moral analysis of computers but on the fact that advances in software have made this world possible.

Against this background it might prove difficult to argue *against* the need of studying the software industry. 'The software industry' is, however, a rather large concept, containing enough material to fill a whole library. This thesis is only one book in that imaginary library, but, on the other hand, it is dedicated to one of the basic moments of the software industry: the experience of writing a program is, at the end of the day, the one in which the whole thing originates.

At the heart of the programming process, surrounded by deadlines, marketing campaigns, investment assessments, political struggles, organisation charts, beta releases, testing teams and numerous other things lies the phenomenon that we are going to focus on here: the programmers' experience of writing software.

All the systems described above, and so many more that I have not mentioned, have once been designed and coded by programmers. And are right now, as you read this, being maintained: a new functionality must be added, or someone has found yet another bug that must be fixed, or the application must be ported to different hardware, or the system needs to be restarted, or someone managed to delete something that should not have been touched… constructing and keeping this software world running is a huge enterprise which consists, at its most elemental level, of the meeting between programmers and code.

This meeting takes many forms: a programmer creating new code, or reading, modifying, discarding, admiring, despising and/or damaging already existing code, or a programmer discussing programming issues with other colleagues, trying to solve a problem, commenting on new tools, criticising someone else's work… All this is part of the programming experience, the nucleus of this thesis, and will go here under several names: personal relationship to one's code, the programming experience or, simply, programming. At any rate, and whatever we choose to call it, this 'programming', can be approached from two different perspectives, which we may label 'public' and 'private'.

From the **public** perspective, programming consists of measuring and calculating, or, more generally, of applying one's knowledge to finding the correct solution to a problem. From this perspective, the meeting between programmers and software is governed by some sort of objective methodology according to which programmers simply carry out calculations and write in the correct commands. These calculations may be quite complex but they are calculations nevertheless: according to the public approach programming does not allow personal expression, it is simply a matter of solving mechanic puzzles.

Hence, programming is regarded as an *exclusively* instrumental activity, something done just in order to achieve a result: the actual *doing* is meaningless. Consequently, programming becomes a dry, soulless activity, something better optimised, minimised, made efficient… something that should disappear. If machines could do it, so much the better: programming in itself is not interesting, only the results are. Well, actually, only the *utility* of the results has significance. The result itself, i.e. the code, is also perfectly uninteresting. The public, utilitarian perspective reduces the meeting between programmers and software to a mere functional step in the machinery of the software industry. The inputs are time, money and technical specifications; the process is called programming (and should – and can – be optimised), the output applications.

Marshall Sahlins suggests further that this utilitarian perspective is not only public but also legitimate, since we live under the assumption of scarcity (Sahlins 1972). This assumption is Sahlins' way to describe the Western economic attitude:

That sentence of "life at hard labor" was passed uniquely upon us. Scarcity is the judgement decreed by our economy – so also the axiom of our Economics: the application of scarce means against alternative ends to derive the most satisfaction possible under the circumstances. (:4)

With such an economic attitude at the base of our society, it is understandable that the natural way to relate to software production is through the concept of efficiency. Programming, in itself, does not ease our impression of lack, only programs (products) do. Hence, the activity of programming is devoid of public value and the more efficiently software is produced, the better. Consequently, much of the literature dedicated to programming, both on the popular management and on the academic side, is concerned with the problem of making the personal aspects of programming disappear: the ultimate goal seems to be to devise models to make of it an activity as mechanical and optimised as possible.

I am aware that this description has a certain moralistic tone to it, and that I seem to be saying that an exclusively instrumental perspective on programming is immoral. My opinions on this matter are, however, perfectly uninteresting. I have no intention of undertaking a moral analysis of software development, and the goal of this thesis is *absolutely not* to prove that such a perspective is morally abusing. When I say that it is reductive, I only mean that such a perspective does not account for a number of evidently existing phenomena (which I shall present as we go along), not that it would be dehumanising.

This perspective is based on the assumption that programming is an objective activity. And even though it is a commonly held assumption, it is incorrect (as this thesis will argue). Engineering journals (IEEE Software, ACM Transactions on Office Information Systems, Communications of the ACM, Science of Computer Programming, etc.) are dedicated to the publication of articles about how to make programming more efficient, studying (and measuring), among other things, the effects of different programming alternatives, the way in which knowledge is transmitted from experts to novices, the functionalities made possible by new programming languages, etc. There are also a number of books about the management of software projects, some of them suggesting ways to assure that deadlines are met (McConnell 1996) (Yourdon 1997), others lay out methodologies to obtain software of higher quality (Hunt and Thomas 2000) (Cooper 1999) (Gabriel 1996) (Gancarz 1995), and others explain how to apply the lessons learnt in other fields (Winograd 1996). In all these cases, programming is considered a purely instrumental activity, a process that should be optimised. Such a view deprives code of any intrinsic value, reducing it to a tool, and making the phenomena presented in this paper at best a illegitimate and at worst invisible.

Naturally, it is perfectly reasonable to elaborate suggestions for making the programming effort more efficient, more 'plannable' and more controllable. The question is not whether such a approach is legitimate but whether one can expect it to be fruitful.

This approach is legitimate because it is clear that software development projects, the innermost engines of the software industry, have an erratic nature. Legend has it that no programming project has ever been finished in time, or within budget, but we needn't go to such extremes to admit that many applications cost more than planned and arrive late.

The effectiveness of the approach will depend on whether it manages to address the reasons behind that erratic nature. Judging by the assumptions it makes about the activity of programming, this approach locates the origin of the erratic nature in a lack of proper project management and planning techniques. Or in an insufficient amount of measurements (in a behavioural / Taylorist spirit). For all the critique it has so far received here, it must be said that these problems are ingenuously tackled by the above listed literature, and that it contains a number of interesting insights about the nature of software development projects. And a number of useful recommendations too.

It is also possible to account for the erratic nature of programming projects in the peculiar nature of software. Frederick Brooks does this in his seminal *The Mythical Man-Month* (Brooks 1995) where he discusses "why programming is so difficult", arguing that software has a special nature, being both perfectly logical and perfectly immaterial. This, Brooks says, makes programming an activity like no-one else, an activity that we still have not learnt to master. Brooks' book is meant to help programming managers to understand the effects that the special nature of software has on the progress of software development projects.

The Mythical Man-Month, despite mentioning some of the phenomena that will be presented here, does not really examine the possibility that the erratic nature of software projects may originate in the personal relationship programmers establish with their code. In other words, it does not consider the alternative that the problems to meet deadlines may derive from programmers working too hard, rather than from (the more obvious argument) them not working hard enough. Programming for them is an act of creation, and their code a mirror of their identity: something that speaks about them and that cannot be put together in haste. In the following chapters we shall study how programmers relate to their work, and we shall see that this relationship gives raise to phenomena that cannot be explained from the public perspective. In other words, we shall study the private aspects of programming.

From the **private** perspective, programming is not the unimportant activity that must take place in order to obtain useful applications, it is something with intrinsic value. While from the public perspective, the meeting between the programmer and the software is simply a functional one, the private view regards it as a fundamental part of the entire process of software development. Marketing, distribution, corporate policies, investments, salaries, expected returns and other details obviously exist, but they are secondary.

The programmers' understanding of programming is *partly*[2] based on this perspective, even if, judging from some of their discussions, understanding programming from this perspective is essential to being a 'true' programmer. Programmers are seldom fanatically focused on their code, there is a whole spectrum of attitudes to the private aspects. Some programmers are almost oblivious to public concerns such as usefulness (not function) and costs, focusing on the intrinsic qualities of code, whereas some are almost oblivious to the intrinsic qualities of their software, focusing on its cost and usefulness. Only very extreme hackers are blind to the public aspects of software (profitability, usefulness, etc.), but for most programmers, code is something important (with varying intensity) in itself, regardless of its role in a wider context.

For them, programming is a meaningful activity in itself and code should be elegant. What uses the code might be put to, or what role it fills in the corporate strategy, or how much it will cost are also meaningful aspects, but not the only measures of its worth. Programmers are also concerned with creating code that speaks well about its author. Code is, unlike in the public case, not merely what makes the processor run, creating an application that can be used to offer services (a word processor, for instance); code is also a sign of its creator's skills and preferences.

From close up, as we shall see, programming does not consist of calculating the right commands. In fact, there cannot be 'right' commands since there are a number a different alternatives which all provide the same functionality. This has allowed for the appearance of different programming styles, indeed of different fashions. What from the distance of the public perspective, always at arms length from the meeting between the programmer and the code, are objective technical decisions turn out to be, when observed more carefully, subjective choices based on a mixture of aesthetic preferences and beliefs. Programming is something personal, and we may understand some of its aspects better by comparing it to a religious ceremony instead of thinking about it as a matter of calculations.

These aspects are here called 'the private aspects of programming', and they are a third origin of the erratic nature of programming projects. This thesis is dedicated to the study of those aspects, with, among others, the aim of offering programming managers one more perspective from which to understand the dynamics of software development projects. So, software managers not only have to face the complicated mechanisms characteristic of projects, or the difficulties inherent to software, they also need to take into account the programmers' relationship to code. In other words, they need to understand the difference between solving computing problems and programming.

This difference lies at the heart of the concept of private aspects of programming and can be explained as follows. Imagine that a programmer is told to write an application (a program) that allows a company to store all the data related to their sales, their customers and their providers. The client simply wants a database with all the data and a program that can show them how much they have sold to any particular customer, or of any particular kind of product, or a given month, or any combination of the above. The programmer has go to the client's office, be taught the information relevant to the project (how the client classifies the products, how often new data arrives, who is to feed it into the database, and how much this person should know, who accesses the database and what they need to know) Let's imagine that, after a while, she knows exactly what she is supposed to do. In other words, that she has defined the computational problem, which is to write the set of instructions that will make the computer store and present the desired information. The difference between solving this computational problem and programming the application (and between "set of instructions" and "program") lies in the programmers' personal involvement. It lies, in fact, in the private aspects of programming. When I say 'to solve a computational problem' I mean 'to program', but in the sense presented by the public perspective. In this sense the programming happens without the programmer thinking about whether the program is beautiful, or something to be proud of, or something that reflects her preferences. Solving computational problems is a detached activity, a sort of calculation carried out without engagement, a process of perfectly objective reasoning. Programming, on the other hand, is a form of expression.

From the public perspective, programmers solve computational problems. They sit down with a set of technical specifications and then try to meet them, in a rather straightforward and unproblematic manner. This notion, it is my impression, does not originate in careful observations of the programmers at work, but perhaps in the, economically and morally respectable idea that, regardless of how programming is actually carried out, this is *how it should be done*. From this perspective, let me insist, the process of software creation has no intrinsic worth, and hence should be mastered, mechanised and optimised in order to produce more with less (remember

Sahlins's assumption of scarcity). Hence, scholars and consultants are dedicated to bringing programming projects under control, to make them predictable.

This is understandable, even if perhaps not as rewarding as they might expect. From the private perspective, programming is above all a creative activity and mixes badly with order and predictability. Not because 'creative activities' would require bohemian conditions, even if some programmers claim that the nine to five kind of job severely hinders their creativity, but because they involve people in more complex ways than just calculating activities do. One's programs are personal, and discussions about them are personal, and the technical decisions are personal, and the managers' or the colleagues' intrusions and commentaries are taken personally. Writing software is a way of self-expression, which in turn means that the management of programming projects is also the management of individual personalities.

MANAGEMENT OF PROGRAMMING PROJECTS

This work has been written at the Department of Industrial Economy and Management of the Royal Institute of Technology (Stockholm), and it is from the point of view of the Industrial Economy field that it must be understood. The traditional 'industrial economic' approach originates in the work of F.W. Taylor and his 'Scientific Management' methodology (Taylor 1967). Even nowadays, much of what is written about programming (as a manageable activity) can be directly connected to those ideas from the beginning of the last century.

Taylor's work is, however, not part of a Management Science, it is a description of a management methodology, and not a study of what organisations, or management, or work are about. Still, his ideas continue to have a major influence on academics' approach to the subject even today. Mainstream researchers in the field focus on the definition and measurement of the phases of a software project (for instance by clocking the time spent in different activities), with the hope (I assume) of finding the key to the regularities that will allow them to write the formulas that describe the programming effort. This approach is based on the assumption that programming is not a personal process but a mechanical one (or at any rate, more substantially mechanical than personal), like the falling of a stone, and that it will show the same uniformity and stability as this latter.

By showing the importance of the personal aspects of programming, this thesis exposes the limits to what can result from such a Taylorist approach, and in doing so I present here a distant relative of the Hawthorne experiments. As I said, there are interesting insights to gain from the mainstream literature and research publications. It is indeed possible to increase the programmers' productivity as the result of a careful statistical study of their work, or as a result of a stricter project management. But one should be aware that programming, perhaps even less so than assembling relays, is not an automatic process, if by 'automatic' we mean devoid of human aspects[3]. Each program is different, and constructing a useful model of the programming process may well be as impossible as developing a theoretical account of the painters' efforts. In other words, programming is a personal, creative activity, and the gains that can be achieved through measurement and optimisation are limited. Exactly how limited, it is impossible to state categorically but I am convinced that the Taylorist approach is not as rewarding as many academic texts seem to assume. The programming effort is, in many important senses, more creative than mechanical, and there are few significant regularities to be found, and few gains to make from those.

The approach here is hence not to find statistical regularities with which to construct a Science of Management but to explain what programming is. The Science, at this stage, consists of the careful observation and description of not only actions (which can be clocked, in a behavioural manner) but also of experiences. The goal is not to develop formulas to apply but to offer explanations, a goal that assumes of course that managing is not (and cannot be) the application of formulas but the art of manoeuvring beliefs, intuitions, guesses, wills, expectations, skills, time, money and other resources (see – regrettably, only Swedish readers – Gustafsson's *Känsla för Zap* (Gustafsson 2000)).

I am, naturally, not the only one who holds such an opinion. One of my most notable predecessors is the above introduced Frederick P. Brooks Jr., himself a software manager (later Kenan Professor of Computer Science at the University of North Carolina at Chapel Hill), who wrote one of the most influential books on the subject of software development management: *The Mythical Man-Month* (Brooks 1995). In it, he reflects on his "very educational, albeit [] very frustrating" (:x) experience of managing the development of the legendary IBM Operating System/360, in direct response to "Tom Watson's [IBM's general manager at the time of the project (1964-65)] probing question as to why programming is hard to manage" (:xi). Although he does include some statistical reports, the book is actually an essay, in which he tries to explain "why programming is hard to manage", i.e., why programming projects run so late and so over-budget.

The software was written, and hard to manage, around 1965, and judging by the date of first publication

(1975), one can only assume that programming had not become much easier to manage ten years later. A decade on – 1985 – he presented the article *No Silver Bullet - Essence and Accidents of Software Engineering* (included in the referred edition of *The Mythical Man-Month*) where he states that "there is no single software engineering development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity" (Brooks 1995) (:179). In other words, Brooks was suggesting that programming was still hard to manage. In the 1995 edition of *The Mythical Man-Month*, he included an extra chapter to answer the criticism set forth by that bold statement (Chapter 17). Programming was still hard to manage, and I see nothing that would have changed that in the last eight years.

Brooks explanation as to why this is the case has to do, as I mentioned earlier, with the inherent properties of software: complexity, conformity, changeability and invisibility. We need not comment them further here. Suffice to say that Brooks believes that the complexity of software systems (aggravated by their need to conform to arbitrary human institutions, the changing environments in which they must be written, installed and used and the impossibility to visualise them) makes programming and its management especially hard. What he proposes is an explanation of the peculiar difficulties of programming, and not the finding of regularities. Managers who read his book are supposed to gain insights, not to learn applicable formulas.

My explanation of why software development management is difficult has to do with another important characteristic of the programming effort: the fundamental role that creativity plays in it. Its most important consequence is that it makes of programming a personal adventure, and a matter of self-expression. We shall see how personal styles appear at all levels of the programming effort: from those held in less esteem (coding) to those considered truly creative (software design).

We are used to consider software as a tool, and we usually come in contact with it under the form of an application. For instance, we see the user-interface and through it we interact with an invisible system. With a machine indeed, if a virtual such: pushing a button moves some mechanisms inside that produce the result we expect. For instance, we dial our friend's number on the phone, finish it with a "yes" and a huge system of virtual cogs and shafts and pistons allows us to speak to her.

Programmers, on the other hand, work with these cogs, and the system is not at all invisible (even if it cannot be visualised). It is coded in programming languages, since not even programmers are able to see the electrons being shuffled around in the microchips, but it is absolutely real, and it provokes many different kinds of feelings. For programmers, with a private perspective on software, programs are not only tools but creations, artefacts with many more characteristics than just their price, their function and their user-friendliness. They can see whether a piece of code is "clean", or if it is "a horrible mound of spaghetti", or if it shows "elegant and masterful design", and, most importantly, this makes a difference to them (more to some than to others). It should also make a difference to us because the way in which programmers relate to software has important consequences in the way they create (and maintain) software. It should, at any rate, make a difference to those in charge of managing, financing or buying programming projects.

Code, despite its invisibility (if the application runs well, no-one should ever need to see it; in the case of commercial software, you are not allowed to see it regardless), carries lots of connotations. It speaks about the overall structure of the application, the conception of the problem, the assumptions about the final user, the large and small technical decisions that have been made, etc.; in other words, it speaks of the comprehension, the aesthetic preferences, the skills and the ingenuity of the author. Given all this, it is perfectly natural to assume that programming can be experienced as much more than just solving the user's needs. It can be easily interpreted as a symbolic activity through which to express one's view of the world.

The idea, as the reader will see, is to present a description as thick as possible of the private aspects of programming (or, in other words, of the visible consequences of programming being creative and personal). I have chosen, in view of the empirical material that I gained access to, to focus the description on the aesthetic aspects of programming. We shall see what a beautiful program is, and how important the concept of software aesthetics is for programmers. From this focus, we shall move to the surrounding subjects (vanity, heated clashes, admiration, despise…) and slowly, a picture will emerge of programming as a personal form of expression that can be described as a religion. The programming effort turns out to contain phenomena that can be successfully interpreted as rituals, beliefs and sacrifices (if a bit idiosyncratic). Clearly, software development management involves much more than just setting deadlines and calculating the resources needed.

The Mythical Man-Month is an answer to the question 'Why is programming so difficult?', but only a partial one. Brooks focuses on the aspects inherent to software (to the essential tasks of programming) that make it difficult to develop (and to manage its development). By concentrating on the characteristics of software, he does not take on board the fact that it is humans that write it (a fact that also makes programming difficult), perhaps because this issue would have led to a book twice as long, perhaps because the issue did was not quite ‘scientific’, or perhaps because he thought that this is an issue better dealt with on a personal level. It is therefore surprising that *The Mythical Man-Month* starts with an enumeration of the joys of programming:

> The craft of programming “gratifies creative longings built deep within us and delights sensibilities we have in common with all men,” providing five kinds of joys:
> - The joy of making things
> - The joy of making things that are useful to other people
> - The fascination of fashioning puzzle-like objects of interlocking moving parts
> - The joy of always learning, of a nonrepeating task
> - The delight of working in a medium so tractable – pure thought stuff – which nevertheless exists, moves, and works in ways that word-objects do not.[4]

Brooks, for unknown reasons, does not grant this joy and its consequences, any greater role in the subject of programming management. This thesis fills in that gap, answering the question ‘Why is the management of software development projects so difficult?’ from another perspective.

1 The words 'program' and 'application' are not exactly synonyms but will be used interchangeably throughout the book. The main difference between them is that an application generally has rather well defined function, whereas programs may have more vague uses. For instance, an operative system can be called a program, but not really an application. Simply put, all applications are programs, but not all programs are applications. These two words will be accompanied by two other, used as synonymous: 'software' and 'code'. Strictly speaking, the code of a program is its set of instructions, whereas the program is the virtual machine created by the code running on a computer, but programmers, and to a certain measure I too, use both interchangeably. Software is generally used to refer to the body of programs 'out there', but will be used here very much like 'program' or 'code'
2 I cannot emphasise this enough: most programmers are very concerned indeed about the usefulness of their code. This thesis will take this for granted, and focus instead on their concern for code itself, but it is not my intention at all to suggest that this concern is the essence of programming. It is only a part of it, even if it is the only one dealt with here
3 I cannot here but recommend, to the Swedish speaking reader, Johan Asplund's revisiting of the experiments (see chapter 17 in Asplund 1987)
4 This neat summary is his own. In the anniversary edition of The Mythical Man-Month he includes a new chapter at the end with a summary of the statements it contains. This statement (1.2) can be found on page 230

II

Method and Empirical Material

A short presentation of the author's motivations and of the origins of the study. An argument about its similarities and differences to Science and Technology Studies and about its ethnographical ambitions and shortcomings. Finally an introduction to the nature of the empirical material.

This thesis is about programming. I enjoy programming, I have done ever since my parents bought me a C64, and I have even been a professional programmer for a short period of time (although I never worked on particularly advanced projects). Lately I have not programmed much, instead I have been reflecting on the many aspects of the activity of programming, and this book is the result of that interest. Inspired by Johan Asplund's *Om undran inför samhället* (Asplund 1970), my approach to the subject of this thesis has therefore been led more by curiosity than by theoretical considerations. I simply wanted to examine programming as an act of creation.

In this sense, this thesis has things in common with the academic effort called 'Science and Technology Studies' (STS), a research field interested in bringing to light human, as opposed to purely deductive, processes that take place when science and technology are created. The STS researches take us to laboratories, departments and ateliers to show us, in all their richness, the multifaceted processes that go on simultaneously in those factories of science and technology. They relate the underlying assumptions, the political struggles between team members, the changing economic conditions, the legal and administrative settings; in short, all manner of activities, public and private, that finally result in scientific propositions or in technological artefacts.

Bruno Latour, one of the leading voices in this field, wrote a book with a fitting title: *Science in Action* (Latour 1987). As much as *Science in Action* is about the private aspects of the production of science (the bits which do not quite fit with the idealised methodological descriptions), this thesis could have been titled *Programming in Action*. However, Latour is more interested in the process by which politicians, scientists and engineers enlist actors (artefacts and people) to make their claims become accepted, and in a sense, true. Even if he also discusses, in

Feyerabend's spirit, the anarchic procedures of scientific progress, he does not write about its personal components (for instance the aesthetic aspects), and I have therefore decided not to use that title.

There is another meaningful difference between this thesis and the 'normal' STS approach: I do not take you, the reader, to the places where programming is done. I have been to a few such places, even if I have never conducted any ethnographical study worth the name, but I have decided that the most interesting action, for the purposes of my argument, takes place not where programming happens but where programming is discussed. After all, this is where programmers relate their experiences, and I am much more interested in that than in a study of their (external) acts. Hence, this thesis is not exactly about the activity of programming itself, but about what programming *means* to those who do it. Also formulated, throughout the text, as "about the relationship they establish with their creations".

These discussions about programming are held a little bit everywhere, including, naturally, the same places where programming happens. While working, programmers discuss what they are doing, both under formal sets (update meetings and so on) and at informal venues (while eating lunch, for instance). There is, no doubt, a lot of interesting insights to be gained by studying this sort of exchanges but, for reasons that will be explained later, I chose something else.

To speak about one's work at work is not something exclusive to programmers, I would imagine most people do. But there is another place where programming (among other things) is constantly discussed, a place in fact created by programmers: the internet. Most of the empirical material that will be presented here has been gathered from internet fora, mostly from the programming forum Slashdot (www.slashdot.org). But the phenomena that support the argument are not exclusive to internet, they can be found on basically every programming environment: programming books, schools, conferences, etc. However, as we shall see, the internet, and in particular Slashdot, offer the researcher considerable advantages over other forms of communication.

WHAT KIND OF STUDY?

I have studied discussions on the internet, and the findings from there have been corroborated by other observations, including literature, interviews and, well, personal experience. It would not be incorrect to say that it happened the other way round: that observations from the literature, interviews and personal experience triggered a detailed study of internet discussions. Now, what kind of methodology is this, studying discussions on the internet? It is definitely not an ethnography, since I have not been dwelling among my study subjects. Neither is it a virtual ethnography, where the researcher takes part in a virtual (internet based) world. Slashdot is a virtual culture, but I have not studied it as such. In fact, I have not studied the cultural patterns of Slashdot at all, I have only studied what was said in two of its discussions.

The basic methodology used here is text analysis, more specifically, grounded text analysis, since the fundamental conclusion (the existence of private aspects of programming) does not only rest on the study of Slashdot comments but also on, as I just mentioned, interviews and personal experience. The analysis of the Slashdot comments did not contain any oddities: I printed out the messages, and read them several times, comparing what was said there with what is said elsewhere (literature, interviews). Driven by curiosity, and not by a hypothesis, I engaged in a classic hermeneutical circle, interpreting my material, gaining new insights, interpreting it again, and so forth. Slowly I started to see ways of ordering the data, possible classifications, really; the most important of them dealt with programming aesthetic ideals and with attitudes towards the issue of software aesthetics (they are presented in coming chapters). What took me most time, however, was to see that all these different opinions were manifestations of the same phenomenon, namely the private aspects of programming. So the primary conclusion came last, and the argument was not clear to me until then.

So there is little to tell about formal methodologies, due to my lack of original hypothesis, I have found myself gathering material somewhat anarchically (reading books, visiting programming fora, interviewing/chatting to programmers). I only knew what I was looking for in a vague sense, but found it clearly exposed in my data once I stopped to analyse it carefully. The main source is are two Slashdot discussions, but I think it is better to start with a short outline of the secondary sources: my personal experience, the literature and the interviews.

The role of my personal experience in this thesis is unclear but quite decisive, and hence, worth bringing up. Now, considering the skills of the programmers that will appear in this thesis (both those that took part in the Slashdot discussions and those legends that will be presented), I would like to insist that my programming knowledge was never remarkable, I was never a hacker. Some of the technical details that came up in the discussions or in the literature (consider, for instance, (Bentley 1986)) I only have an vague understanding of, and some others I do not understand at all. But I do know enough to be able to understand what they are speaking about, even if I could not program at that level. This knowledge has been essential in the writing of this thesis, but my goal has been that the argument should be understandable with the help of just some programming basics. Therefore, I often stop the technical descriptions with a "we do not need to know this", or something of the kind.

Back when I was a programmer, I never stopped to think about private and public aspects, at least not in a distinctive way. We all understand that some explanations are legitimate and some are not (for instance, we could not quite defend using Delphi with the argument that we liked it better), but what I liked to do was to program, not to analyse my programming. So the baggage that I bring with me from those days is mostly composed of blurred memories of discussions and a strong feeling that the code was important to us in other ways than just its final function. This feeling, of course, forms part of the fundaments of this thesis, even if, by itself, it would never have amounted to much.

At the origin, this working name of my PhD project was *Technique and Aesthetics* (Teknik och Estetik), and for some unfathomable reason I did not quite associate technique with programming. Only after a couple of years did I eventually see the connection, see the way I should (could) go, and focus on programming and programmers. Had I been a structural engineer instead, I would probably have written about that instead, so, in this sense, the fact that I was a programmer has had a definite influence on this thesis.

There is a huge body of literature that deals with programming, but little of it is dedicated to its private aspects. The mainstream approach is, as I outlined previously, to treat it as an activity that must be controlled and made efficient. This does not prevent some manifestations of private aspects to appear (see, most notably (Weinberg

1971), (Gelernter 1998) and (Dahlbom and Mathiassen 1993)) but they do not allow a proper study of the phenomenon.

There are some texts, really not many, that recount programming experiences. The most interesting one I have found written by a programmer, about her own professional experiences, is *Close to the Machine* (Ullman 1997), in which Ellen Ullman tells us about life as a professional programmer (and other assorted issues). She does this in a descriptive style, without the normative ambitions otherwise so common (it would seem that almost everyone who writes about software wants to convince us that s/he knows how it should be done). Moreover, the book is written in a personal style, for which I am naturally very thankful. She has written some other articles which are also highly recommendable for anyone with an interest in a programmer's perspective on computers and software (Ullman 1995; Ullman 1998).

Apart from that first-person account, there are also a few books that tell other people's stories. Tracy Kidder's *The Soul of a New Machine* (Kidder 1981) tells the story of the creation of Data General Corporation's 'Eagle', or Eclipse MV/8000, a revolutionary computer, including sections about its software. His detailed descriptions include interesting quotes by programmers (and hardware designers) about their personal relationship with their creations. Levy's *Hackers* (Levy 1984) is the classic account of both famous and obscure – both kinds totally dedicated – programmers; his interviews let us in on their experiences of software writing. In fact, any book that covers the relatively short history of software is bound to include comments about what programming means for programmers, even if, most of the times, it is only dealt with tangentially.

Last but not least, I was very impressed with John Bentley's *Programming Pearls* (Bentley 1986), a technically advanced book in which he presents small programs, “programming pearls whose origins lie beyond solid engineering, in the realm of insight and creativity” (from the preface).

The texts listed here are naturally not the only ones, but they are the most influential. All were part of the hermeneutical process, even if the thesis contains few direct references to them.

Regarding interviews, I have to admit that I failed. Not miserably, but nevertheless failed. The immediate alternative when dealing with subjects such as ‘private aspects of programming’ appeared to me to be interviews. Ques-tionnaires and other formalised devices were definitely out of the question, in fact I am afraid that, I must admit again, I did not even try. My intention was to describe and analyse an aspect of programming based on what programmers say to each other, in a ‘natural setting’, not on their answers to surveys of any kind. The first approach was based on open interviews carried out with programmers. I hoped to collect material rich enough to allow some in-depth analysis (a proper thick description (Geertz 1973)), but after a few interviews I realised that would not happen. The interviews were nevertheless very valuable, both as a reaffirmation of my original feeling (that programming is more than just solving computing problems) and as a more grounded introduction to the subject. Looking back on them, I see that they contained more information than I could see.

From a methodological perspective, however, I have come to believe that interviews – although sometimes the only available alternative – are not the best way to learn about what goes on among programmers. I am not the only one to see the problematic nature of interviews, see (Bryman and Nilsson 2002); and I am not totally opposed to them, they can offer valuable complementary information. But could they have been enough in my case?

The idea was to learn how the word 'beautiful' – or 'elegant' – was used among programmers, and possibly one of the least fruitful ways of doing this is by asking them. It is always better to ask them under the form of an open interview than by sending them questionnaires but, no matter how open the interview is, it does not quite work. The main problem, as I see it, is that programmers do not necessarily pay much attention to their use of the word 'beautiful', they just use it. It is unrealistic to expect that they will later on remember how they have been using it. You could say that I was trying to approach a, in Heinz Kohut's terminology[5], experience-near concept (see (Geertz 1993) chapter 3) from an experience-distant perspective. A programmer uses 'beautiful' "effortlessly [...] to define what he or his fellows see, feel, think, imagine, and so on, and readily understands [it] when similarly applied by others" (:57). But my approach towards it was that of a researcher seeking to "forward [my] scientific [...] aims" (ibid). It was unfortunate that both were the same words, but, being my first interviews, it is not so surprising.

I also tried the idea of asking them about other things, hoping that they would start using the b-word on their own, unconsciously, so to speak. Having been a programmer myself, I could discuss some technical details with them. And yes, the word did appear here and there, but I did not manage to make these discussions rich enough. I was definitely to blame, but I also think that the awkwardness of interviews, and the lack of a proper working context had something to do with it. Whatever the reason, I was just not getting enough material for me to get a foothold in it. Or perhaps the material was too subtle for a novice like me; at any rate, I decided to stop quite early in the process (I only interviewed 11 programmers in total, in 5 open interviews of about 1-2 hours each).

What I was missing, I have come to realise, is the richness of *real* situations. I did entertain the idea of working for a programming company for a while, with the intention of writing ethnographical notes all the while: a proper ethnographical study. This is an idea that I still think would be worth carrying out, perhaps as the continuation of this study, but at the moment I postponed it for something else: attending virtual meetings.

All these three sources of empirical data were used, and, to a certain extent, analysed, before I found the two Slashdot discussions. They lacked the richness of real life situations: some because I was a poor interviewer, some because their purpose (literature) was a different one, and some because my memory failed me. But all of them became much more informative once the Slashdot discussions entered the hermeneutical process. But I did not find them at once, it took me some time to get there.

As mentioned in the introduction, the material needed to study the private aspects of programming can be found (at least) in two places. The first one, and most obvious, is where the programming happens. Programmers, while programming, say things about what they have done, about the code they must modify, about their own and other people's ideas… they exchange opinions of all kinds: irreverent, important, silly, insightful, etc. This conversation goes on, so to speak, in the background, while the foreground is occupied by technical documents, official meetings, budget decisions and other legitimate forms of information exchange. Gustafsson calls this "små pratet" (small talk), and urges anyone who wants to understand human enterprising to pay attention to it (Gustafsson 1994).

Now, I did not have access to this exchanges, but I did have my internet connection. Apart from at their working places, programmers' also exchange opinions on the internet. Perhaps it is because programmers work on computers and know them well, or perhaps because they have become accustomed to the limitations that they put on human interaction, or perhaps it is for some other reason, but the fact is that there is no shortage of active public programming fora on the internet. There is at least one for each programming language, and some even about some specific programming feature, with varying levels of bustle. On these sites programmers discuss technical details, they ask for, receive and give assistance, they publish their opinions about new software tools and about how one should program, they take part in programming contests… They converse, not only about programming, but a good deal about it too.

SLASHDOT

One of the most famous fora is Slashdot (www.slashdot.org), a popular gathering place for programmers of all kinds (55 million page views per month, according to OSDN[6]). Slashdot forms part of the legend of the dotcom era: it was started by a enthusiast hacker (Rob Malda, *CmdrTaco*), who thought that a site where to discuss "news for nerds, things that matter" (Slashdot's slogan) would be a neat idea. The site was programmed mainly by him and, launched from his dorm room in 1997. The site was then bought by OSDN (which sells advertisement locations), making a few programmers ridiculously rich, at least for a while.

Things at Slashdot work as follows: editors receive a lot of suggestions from participants about news they considered interesting. From these they choose a few ones (about a dozen per day) and publish them for discussion (presenting them with a short introduction and, more often than not, a link to more comprehensive information). These short presentations appear in the home page of Slashdot, and from here anyone, it is not necessary to be a registered slashdotter (from now on, slashdotter), can comment it. This is done by writing a message (called comment in Slashdot) on the subject and sending it for publication (i.e. clicking the 'submit' button). The messages have all the same structure:

software manager managing bridge architects... (Score:1, Insightful) by **Anonymous Coward** on Tuesday September 04, @04:20PM (#2253011)
manager -> we need to ship this bridge in 3 months.
engineer -> yes, but it's really big and really important
manager -> yes, but it has to ship in 3 months.
engineer -> so how much weight does it need to support?
manager -> i dunno, I'll let you known in 2.9 months.
engineer -> what is it bridging?
manager -> why all these stupid questions, start building.
engineer -> I should do an architectural drawing first.
manager -> why bother, here's some metal, start slapping it together. Remember it ships in 2 months.
engineer -> I thought you said 3 months?
manager-> oh didn't I tell you, we heard a rumour that a competitor will be shipping their bridge in 2.5 months, so we have to beat them.

continue forever.

the reason there is no internal beauty is we (engineers) aren't given any time to build quality (although the argument could be made that the only way to build on schedule is by building quality). The other problem is, bugs actually translate into lucrative support contracts for most enterprise software vendors. Why improve quality? there is no revenue stream there. If users would SUE software development firms (the same way people would sue if a bridge fell when you drove on it) then vendors would suddenly find time in schedules for testing and quality. we do the best we can, given the pressures. My advice, try to learn to say "no" to your manager once in a while, and hire a QA manager with balls who won't let shitty software ship.

The first line contains the title (subject) of the comment, written by the author, and the score, set by the moderators. This figure is an interesting feature of Slashdot discussions. Their function is simply to read the messages that are published and to assign them a score (moderate them). At the origin there were no moderators, only editors, who moderated the messages themselves. As the site grew, the function editor/moderator was separated, but the moderators were still a known group of people (about 400). The site continued to grow and it became impossible to expect these 400 to read all messages and moderate them. *CmdrTaco* then decided to install a complex system of automatic and time-limited selection of moderators (the computer decides, based on different grounds, who should be given moderator status)[7]. The system seems to work satisfactorily, and one could argue that moderation is carried out by the average regular participant.

The scores go from -1 to 5, and signal what the moderator thinks of the comment. These scores are used for filtering: slashdotters can set their preferences so that their computers leave out comments that score less than a certain number. They are also used to decide who can become a moderator and who has right to a higher intial score. This has given rise to a hierarchical structure, since Slashdot keeps a public track of the comments sent in by its registered users, along with their average scoring. Some slashdotters therefore react when their comments are moderated down.

At any rate, I have not attached much importance to the scores, since I, as a researcher, approach the discussion with a different goal than the participant. Hence, I have found interesting opinions on 0-scored messages (and even in -1, but these are rather infrequent).

Another interesting part of the header in the example above is the comment "insightful". This is set by the moderators, who actually moderate a message up by assigning it 'good' characteristics and down with 'bad' characteristics. This is the current list of possibilities offered to a moderator, together with an explanation of how to use them (for new moderators); there is little doubt about which ones are good and which ones bad:

Normal -- This is the default setting attached to every comment when you have moderation privileges. Normally, you should not need to actually select this option, but if your mouse slips and you accidentally moderate up or down a comment you didn't mean to, you can undo that mistake by choosing Normal before you hit the "Moderate" button.
Offtopic -- A comment which has nothing to do with the story it's linked to (song lyrics, obscene ascii art, comments about another topic entirely) is Offtopic.
Flamebait -- Flamebait refers to comments whose sole purpose is to insult and enrage. If someone is not-so-subtly picking a fight (racial insults are a dead giveaway), it's Flamebait.
Troll -- A Troll is similar to Flamebait, but slightly more refined. This is a prank comment intended to provoke indignant (or just confused) responses. A Troll might mix up vital facts or otherwise distort reality, to make other readers react with helpful "corrections." Trolling is the online equivalent of intentionally dialing wrong numbers just to waste other people's time.
Redundant -- Redundant posts are ones which add no new information, but instead take up space with repeating information either in the Slashdot post, the attached links, or lots of previous comments. For instance, some posters cut and paste otherwise legitimate comments in multiple places in the same discussion; the pasted versions are Redundant.
Insightful -- An Insightful statement makes you think, puts a new spin on a given story (or aspect of a story). An analogy you hadn't thought of, or a telling counterexample, are examples of Insightful comments.
Interesting -- If you believe a comment to be Interesting (and it's not mostly Redundant, Offtopic, or otherwise lame), it is.
Informative -- Often comments add new information to explain the circumstances hinted at by a particular story, fill in "The Other Side" of an argument, provide specifications to a product described too vaguely elsewhere, etc. Such comments are Informative.
Funny -- Think of Funny as being a good moderation choice if you actually think the comment *is* funny, not just because it seems intended to be. Not every knock-knock joke is Funny.
Overrated -- Sometimes you'll run into a comment which for whatever reason has been moderated out of proportion -- this probably means several moderators saw it at nearly the same time, thought it was Funny, Insightful etc, and their scores added together exaggerate its relative merit. (A knock-knock joke at +5, Funny) Such a comment is Overrated. It's not knocking the original poster to say so, but it's probably better to spend your mod points on comments which are deserving of being moderated up.
Underrated -- Likewise, some comments get smashed lower than they

perhaps deserve by overzealous moderators. If you moderate a comment as Underrated, you're saying that it deserves to be read by more people than will see it at its current score. As with Overrated, if you can think of a more specific moderation reason, do so -- if a comment has already been moderated with an appropriate label though, and you just want to indicate that it deserves greater visibility, that's what Underrated is for. However, if a comment is labeled with a fitting (negative) label, choosing Underrated isn't such a great idea, because you could end up with contradictions like "+5, Flamebait."[8]

There is certainly an interesting study here to be carried out about the moderation process, but for the purposes of this thesis, I see no reason to confuse the reader with comments and scores. Therefore, I have decided to include neither of them when I present a message as a quote. However, I have avoided messages moderated as “Troll” or as “Flamebait”, of which anyway, there are only a few.

The second line in the header of the message includes the alias of the author (his/her registration name), the hour and date of publication and an identification number, unique for each comment in the archive. In the example above, the alias of the author is *Anonymous Coward* (all aliases will be in italics in the text), this is the alias that Slashdot automatically applies to all messages sent by a non-registered user. If the sender is a registered user (and has logged in), the message will contain one more line:

Did anyone notice... (Score:1)
by **The Slashdolt** on Tuesday September 04, @04:22PM (#2253023)
(User #518657 Info |http://slashdolt.org/)
That in the revision history that this is the 3rd version of this paper in almost 3 years?
So it takes him almost 3 years to write a 10 paragraph essay with some VB code mixed in, and he is telling us we need to do better? Nice example Mr. Author.

The third line of the header (User #518657 Info |http://slashdot.org/) is a link to a page with information about this user: her last messages, her average score, and some other trivia. Sometimes, this link is written with different words ("user journal") but the contents of the personal page are very similar. There is, hence, quite some information in a header that we are not so interested about, and that I have decided to exclude. The previous message would be quoted:

Did anyone notice... by **The Slashdolt** (#2253023)
That in the revision history that this is the 3rd version of this paper in almost 3 years?
So it takes him almost 3 years to write a 10 paragraph essay with some VB code mixed in, and he is telling us we need to do better? Nice example Mr. Author.

No score, no comments on it, no dates and no links to the personal information. Just the title, the author and the identification number, to make it easier to search it in Slashdot's archive.

Now to the dynamics of Slashdot discussions. The first step is, as mentioned above, to post a piece of interesting news. The editors do this. Then, any participant (registered or not) may write the first comment, this is the "first post". The second participant has the choice of either replying the first post, thereby creating a thread, or else writing a comment to the original piece of news. And the rest of participants can do the same: either reply to an existing message or write their own comment to the original piece of news.

One of the most interesting bits of Slashdot discussions are the threads: conversations formed by several 'replying' messages. For instance, someone could reply to the first post, and in her turn receive a reply from another participant and so on. In the example below (taken from another discussion[9]) we see how this is exactly what happened: the first post, written by *jimmyCarter* is replied by an *Anonymous Coward*, who is in turn replied by another… The mechanics are very simple: you simply have to click on the [Reply to this] link and your message will be published as a reply. The following is an example from another discussion:

Haha, nice save! (Score:4, Funny)
by 2nd Post! (213333) <louis_wang@NOspAm.hp.com> on Friday October 10, @12:45PM (#7184186)
(http://nekobox.org/~sillyoldbear)
By coincidence he also didn't figure out he didn't have much chance of winning *anything*, financial or otherwise, did he?

[Reply to This]

> **Re:Haha, nice save!** (Score:1)
> by zamokzam (218601) on Friday October 10, @12:49PM (#7184226)
> What he's figured out is that it is defamatory to accuse someone of a felony if he hasn't committed one.
> Even with his half-hearted retraction, the damage is done. The student should sue him for his net worth.
> Z
>
> [Reply to This | Parent]
>
> > **Re:Haha, nice save!** (Score:0)
> > by Anonymous Coward on Friday October 10, @12:59PM (#7184347)
> > Oh yeah, and that's tons better than what SunnComm did to him.
> > geez, really, give an inch, take a mile.
> >
> > [Reply to This | Parent]

FIGURE, Example of a thread (from a different discussion)

The subsequent replies have the "Re:" attached at the beginning of the original title (even if it is possible to change that) and are presented in a cascading layout, as shown in the figure. A discussion may contain any number of threads, which can be from two messages up to as many as the participants feel like (in some cases all the discussion is only one thread). But often messages are not replied to at all, being left on their own (which does not mean that they cannot be mentioned in other messages).

It is difficult to know whether authors read all the messages posted previous to their contribution, but most likely they do not, particularly since they can filter away all those who have less than a certain score or those moderated as “flamebait”, for instance. Discussions are, at any rate, somewhat discontinuous, with threads being sometimes left at the middle of an ongoing dispute and new, unrelated, comments to the original piece of news popping up. A Slashdot discussion can therefore be described as a bouncing combination of conversations (threads) and more or less stand-alone opinions on a given subject.

One of the good things about Slashdot, from a researcher’s perspective, is that they keep an archive with discussions that have been closed. This archive is freely accessible to anyone, but it is impossible to send in any new comments, so the data is simply crystalline. So there I was, carrying rather unsystematic google searchs on the internet, looking for programmers discussing software beauty, when I stumbled across the following two discussions:

Where Can I Find Beautiful Code?
Posted by **michael** on Wednesday January 24, @09:45PM
from the not-in-anything-I-write-that's-for-sure dept.
eGabriel writes "One of the benefits of free software that I haven't seen explored here is that of the opportunity to study elegant, masterful code. Besides the fact that we can all share and enjoy applications, and reuse their source code, we can also simply download the code and view it for pleasure, to learn from masters of the art. Certainly there are different criteria for determining what makes a piece of code excellent or beautiful, and I am not as interested in discussing that. If however, anyone has found a piece of free software that serves as an excellent example for study because of qualities they as programmers hold dear, I would love to read that code also and be educated thereby. Equally interesting would be code that really is bad, as long as it didn't turn

into direct attacks upon the programmers involved (they can't all be gems!) Any code that shows elegant and masterful design would make for excellent reading; the language in which it is written isn't as much a concern. 'Literate' code is a bonus."

Software Aesthetics
Posted by **michael** on Tuesday September 04, @03:48PM
from the when-tidy-is-not-sufficient dept.
cconnell writes: "Most software design is lousy. Most software is so bad, in fact, that if it were a bridge, no one in his or her right mind would walk across it. If it were a house, we would be afraid to enter. The only reason we (software engineers) get away with this scam is the general public cannot see inside of software systems. If software design were as visible as a bridge or house, we would be hiding our heads in shame. This article is a challenge to engineers, managers, executives and software users (which is everyone) to raise our standards about software. We should expect the same level of quality and performance in software we demand in physical construction. Instead of trying to create software that works in a minimal sense, we should be creating software that has internal beauty." We had a good discussion on a related topic half a year ago.

Imagine my joy. Besides, even if some of Slashdot's discussions do not create much attention – gathering a few participants and resulting in 60-70 comments – these two were quite popular: Where Can I Find Beautiful Code added up to 371 comments, and Software Aesthetics to 748[10]. So there were lots of opinions, praise and insults, offered to anyone who wanted to read. I hurried to save both discussions on my computer and to print them out just in case, but well into the summer of 2003, they are still there.

The kind of data available in Slashdot is, from a researcher perspective, of very high quality. It is, clearly, naturally occurring data: the programmers were not answering a researcher's questions, they were not even speaking to each other in the presence of a researcher; they were simply exchanging opinions. Nevertheless, there are at least two issues that I must consider: one deals with research-ethics, the other with the possibility of slashdotters being untruthful.

Much has been said about the ethics of on-line research, most of it of interest. The fact is that the anonymity that applies to the participants applies also to the researchers: they do not need to present themselves and can carry out their research undercover. Furthermore, the participants' anonymity is only relative. Not only is it, in some cases, possible to trace the real person behind an alias but also, and perhaps more importantly, on-line persona are sometimes as private as real-life persona. Hence, participants may experience a close observation of their on-line conversations as intruding as that of their real life. There are, however, two different kinds of on-line conversations: one is based on real-time chat, the other on publication.

The first one is very similar to real-life conversation: participants interact in real-time, writing one line (or the equivalent) at the time, waiting for the others to reply and replying in their turn. Furthermore, the material carrier of the replies is also ephemeral, although not quite as ephemeral as sound-waves: as the conversation moves on, the letters that formed the reply disappear from the screen (see (Turkle 1997) for a more detailed analysis). All this makes of on-line chatting (including role-playing and other similar forms (Pargman 2000)) an activity that must be treated as carefully as real-life conversation. Hence, in general, one should not 'tape' them (the correct terminology is "save the logs") without consent, neither should one use the names of the participants without asking for permission. There are, naturally, different approaches to this question (Paccagnella 1997) but there is a strong case for treating on-line logs as taped material. This thesis, however, does not contain any material obtained from on-line chatting.

The second kind of on-line conversations is, in one essential aspect, very different to chatting. In fact, it shares more with the newspapers sections of 'letters to the editor' than with real-life conversations. Due to the speed with which messages and articles can be published on the internet, these exchanges are quite immediate, even if they are never as direct as on-line chatting. let alone real-life conversations. But it is not the lack of immediacy that makes them essentially different to chatting, it is the fact that to submit a message to the conversation is to publish it for the general public to read. Slashdot, for instance, is a site where news and articles are commented, and it is these commentaries that make it so popular. Most of the readers of our discussions did not submit anything[11], they were only interested in reading the published opinions. In fact, one needs neither to be a registered slashdotter, nor to be there when the discussion takes place to access the messages. This is probably one of the reasons that Slashdot offers the possibility of expressing your opinion without signing it and choosing instead to appear as 'Anonymous Coward.'[12] There is, hence, no need of taping (or the equivalent) anything at all: a Slashdot discussion is a publication, accessible by anyone (with an internet connection) at any time, and it should be treated as such. With this in mind, I have decided to maintain the participants' real alias (and the message id-number); after all, I am presenting their published opinions and it would be wrong not to acknowledge their authorship. Paccagnella (Paccagnella 1997) reports of similar policies, quoting Rafaeli from (Sudweeks and Rafaeli 1996):

> We view public discourse on CMC as just that: public. Analysis of such content, where individuals', institutions' and lists' identities are shielded, is not subject to 'Human Subject' restraints. Such study is more akin to the study of tombstone epigraphs, graffiti, or letter to the editor. Personal? - yes. Private? - no

Are these opinions always earnest? Or do participants, hiding behind the anonymity of the internet, make things up? In other words, can we trust what they are saying?

This question is not limited to opinions gathered through the internet, you can never be sure that your interviewee is not saying things simply to impress on you, or to go along with you. Neither is it certain that the author of a book has not misunderstood everything, or is not, simply, lying.

The anonymity of the internet brings matters to a head, but the essential method for approaching these questions remains the same. Yes, slashdotters may feel confident to exaggerate, invent, even lie; but what kind of things could we expect them to exaggerate or lie about? And how are those related to the conclusions that I have drawn?

The fundamental message of this thesis is that there exist private aspects of programming, and that these play a role in software development projects. This message is put across with the help of examples of how programmers discuss programming. The argument includes also some classifications, which are perhaps not fundamental but offer a richer description of the phenomenon of private aspects of programming. Finally, at the end of the thesis I sketch a viable analytical toolkit for gaining a broader understanding of the phenomenon.

This fundamental message does not rest only on what I have read on Slashdot, it is a conclusion that draws from a number of sources, including interviews, literature and personal experience. On the other hand, the evidence presented here is mainly based on Slashdot material, and it should be considered with a critical eye. Apparently, the worst-case scenario would be that everyone "trolled", presenting opinions they do not hold just to annoy other participants. Actually, this is only the second-worst case. More calamitous would be that someone found out that programmers responded to *michael's* both callings just to be polite, not because they actually cared; in fact, that they all acted for him.

But such an idea is simply too baroque to be considered.

The second-worst case is also quite unlikely because not only would they need to fool your researcher (this possibility must be considered), they would also need to fool the moderators. And I think it is acceptably safe to assume that a majority of messages (of those not commented as 'trolls') must be earnest, at least in the sense that they show concern for the topic discussed (programming). At any rate, they *sound* earnest, which, for our purposes, is just as good. That the message is not moderated as "troll" proves that a number of programmers find that opinion plausible, and this is more than enough for us, since we are not interested in what each individual slashdotter believes but in the relevance of private aspects for the programming community at large.

Actually, the simple fact that programmers took part in those two discussions proves that there is interest for the subject of software aesthetics. Besides, not one of all the comments denied the existence of software aesthetics (or the fact that some code is more beautiful than other), even if not all of them were fully positive to the idea of writing elegant code, as we shall see.

Some programmers, on the other hand, probably exaggerate when describing the narrow-mindedness of managers, or the incredibly short deadlines they have to work against. Perhaps they even lie about the programming stunts they have managed to pull, but if any of those exaggerations could be proved, the fundamental message of this thesis would only be reinforced: these would be manifestations of the private aspects of programming. To lie about the beauty of one's programs (as opposed to about, for instance, their economic success) must be considered evidence that this aspect is important.

The classifications created in order to analyse the different phenomena presented as evidence (mainly software aesthetics, but also holy wars, vanity, etc.) are more vulnerable to lies. For instance, a classification boundary may be inspired on a message on which a programmer expresses her/his dislike of one-letter variables ("I PUKE!"[13]). If it turned out that this programmer actually likes one-letter variables, and that no-one actually dislikes them, I would have created a useless taxonomy. Hence, on the classifications, I have tried to err on to the safe side, partly not to make claims that I could not defend and partly because the goal is not to produce exhaustive and detailed classifications but to describe how private aspects are manifested. Still, naturally, I firmly believe that the classifications offered make sense, even if I cannot present impeccable evidence for all the details.

In conclusion, there might be some messages quoted here that were not written in earnest (I am almost positive that some of them were exaggerations, and I am sure the reader will recognise them), but the main argument does not hinge on the veracity of each and every message. It relies much more on the fact that the messages were written at all, and that they were not written for the benefit of a researcher.

VARIETIES OF PROGRAMMERS

We must address one more point concerning the validity of my empirical material, namely the question whether the programmers whose voice is brought forward here are representative of the whole. Particularly when taking into account that most of the examples and quotes are taken from a rather idiosyncratic forum, Slashdot. In fact, it is rather unproblematic to assume that, in many senses, slashdotters are *not* typical programmers. Not only due to their unusual interest for online fora but also because there might not exist such a thing as a typical programmer.

There are programmers who write short programs that go into small chips, and those that take part in the creation of huge systems, with tens of millions of lines. There are programmers dedicated only to the latest technologies, and others that work with older software, which may have been written as much as thirty years ago. There are those who would never do anything illegal, and those with a knack for pirating and cracking. There are those who only program from 9 to 5 and others who basically do nothing else but write software. There are young programmers and old programmers, rich and poor, cultivated and ignorant, graduated and self-taught, skilled and inexperienced, polite and bad-mannered, popular and ostracised, extrovert and shy…

This variety also applies to slashdotters, with one exception: they all share an interest for taking part in (or at least reading) online fora[14]. In our particular case they share, more specifically, an interest for taking part in a discussion that deals with software aesthetics. This does not mean that they all agree programmers should follow aesthetic ideals when writing software, but it means they find it worth their while to discuss what programming is and how it should be done. This, in turn, should imply a certain disposition for the private aspects of programming; in other words, it is arguably so that those who took part in these two Slashdot discussions are likely to have an interest in code itself. The problem is that only an insignificant part of the world's programmers took part in those conversations.

This thesis presents no proper scientific evidence that all programmers would all have something in common, such as, for instance, a concern for software itself. Now, is this a problem?

On the one hand, it is not. This thesis is a study of a particular human phenomenon, and the existence of this phenomenon (concern for software itself, or private aspects of programming) is proved beyond doubt. There are definitely programmers that relate to code in ways that are not included by the public perspective of software. Not only among those who took part in the Slashdot discussions but also among my interviewees and among those who are depicted in the literature. So the main conclusion is, as I see it, amply verified.

On the other hand, it is a bit of a problem. The problem has to do with the extension of the phenomenon, or, in other words, with the relevance of the main conclusion. The fact is that I can neither present evidence that the phenomenon is widely spread nor that it plays a significant role in programming projects in general. I nevertheless would like to suggest, based on my personal experience and on how programmers speak about the issue, that a concern for code itself is widely spread and it does play an important role.

This said, I imagine there are programmers who do not care much about what their code says about them, being only interested in whether it does what it is supposed to. Just for the sake of completeness, we may classify programmers according to the following table, in which I have included some hasty but hopefully clarifying comments:

PROGRAMMERS THAT...	...CARE ABOUT CODE ITSELF	...NOT CARE ABOUT CODE ITSELF
...CARE ABOUT FUNCTIONALITY, PRICE, ETC.	good programmers... but sometimes these two concerns are not compatible.	efficient programmers, who perhaps work near the customer and/or the managers and far from other programmers.
...NOT CARE ABOUT FUNCTIONALITY	vain programmers, who might be of the opinion that users and managers should learn to appreciate the beauty of code.	bad programmers, aka 'newbies' or 'coders'.

The curious thing with this seemingly self-evident classification is that it does not hold in general. We shall see that code that for some programmers is beautiful for others is ugly, and hence, programmers considered good (or at least vain) by some, are for others simply efficient (or newbies). Anyway, this is a light-hearted classification and it has no other goal than to show that there are, in this aspect as well, all kinds of programmers.

Summarising, the phenomenon ‘private aspects of programming’ (concern for the code itself) definitely exists but its consequences are unclear, in a double sense: it is unclear how many programmers are concerned about their code; and it is also unclear how this concern affects their programming[15]. So what is in question here is not the validity of the main conclusion but its practical relevance. On this point, I can only found my suggestions on the impression I have gathered throughout the study: that the private aspects of programming are by no means a marginal phenomenon[16].

This phenomenon is, on the other hand, not so well documented. Partly because it does not concur with the public view of what programming should be, partly because it takes place in the background (these two circumstances are, naturally, interrelated). Hence, the two discussions I have found, in which the personal relationship between programmers and code occupies the centre-stage, are an infrequent occurrence. At the same time, they are, in their uncommon concentration, an invaluable occasion for those interested in the private aspects of programming. Nevertheless, even if those two Slashdot discussions are the primary source of examples, they are not the only ones. Some of the other voices (from the literature, the interviews and other online fora than Slashdot) have also found their way to the text.

One more detail: the term ‘programmer’. Given the wide variety of software kinds[17], and given the centrality of the term ‘programmer’ in this thesis, it may appear strange that I have not yet defined more in detail what is meant by it. And, in fact, I shall not go further than saying that ‘programmer’ is, for the purposes of this thesis, someone who designs and/or writes code (the lower limit going somewhere at the level of programming macros for a spreadsheet), regardless of university degrees, job descriptions, free-time interests or social skills. Trying to refine this definition is not only a hopeless endeavour (as it is with most concepts) but also not particularly interesting. However blurry the boundaries of the concept may be, we all know what is meant by a programmer. Besides, the goal here is not to discuss who deserve that title but what the private aspects of programming are. More tersely expressed: we are not interested in what to call people but in what those who call themselves programmers (or the equivalent) say about programming.

Nevertheless, there are other terms to call the people who write software and we may as well throw in a short overview. Probably the most famous alternative to ‘programmer’ is ‘hacker’, introduced to the general public in the eighties by Levy’s *Hackers* (Levy 1984). Nowadays, it can be used in different senses. Among programmers it is still used to denote the skilled, or at least the very dedicated (see (Himanen 2001)), even if in the media it is more and more used to denote those with an inclination to do illegal stuff[18]: breaking into software systems, writing viruses, breaking the copy-security locks, etc (Taylor 1999) (Rehn 2001). What programmers call ‘hackers’ are called ‘experts’ in the public discourse (programmers that can be up to ten times more productive than the average); and what for the public would be ‘poor’, or ‘unskilled’ programmers, are ‘coders’[19] or, in the poorest examples, ‘lusers’.

Some of those who write software, however, reserve the very word ‘programmer’ for the less competent; and use instead ‘software engineer’ for the good ones. I do not make any distinction, using ‘programmer’ in a skilled-neutral sense, apart from in the chapter beauty and functionality, where I differentiate between true-hackers and software-engineers. However, I do not use these terms exactly as they do, but all this will be explained in due course. The important point to remember is that I shall use ‘programmer’ in a straightforward sense, without any wish to be controversial.

In fact, when analysing the Slashdot discussions, I use indistinctly the terms ‘programmer’, ‘participant’ and ‘slashdotter’; even if not all slashdotters (people who read Slashdot discussions) are programmers, and not all programmers are slashdotters. However, concerning the first point, the opinions that count for this thesis could only be written by programmers (hence, all slashdotters that matter are programmers); concerning the second, I have been careful to use the expression ‘some programmers’ (and slashdotters are ‘some programmers). The aim, at any rate, has been to make the text more easily readable, not to make unsupported claims.

5 In John Van Maanen's terminology (Van Maanen 1979: 539 - 550), I was confusing first-order and second-order concepts.
6 http://advertising.osdn.com/advertising/technology/sites
7 This overview of the moderating process is somewhat simplistic. You can read about it in more detail at
http://slashdot.org/faq/com-mod.shtml
8 http://slashdot.org/faq/com-mod.shtml#cm2500
9 For the curious: Discussions that have been archived – and the two dealt with in this thesis were – do not offer the possibility of replying, they have been closed. The [Reply to This] link is therefore deleted, making these discussions useless as examples of the Slashdot dynamics.
10 It is difficult to know exactly how many slashdotters actually published their opinions, since a number of them used the anonymous option (and it is possible to use different aliases), but I estimate that they may have been around 300
11 This statement is based on simple mathematics: if there are more than 55 million page-views per month, as the owners of Slashdot claim (http://advertising.osdn.com/advertising/technology/sites/), and there are about 12 discussions per day then there should be, in average, about 152.000 hits per discussion. This is not really the number of people reading each discussion, since many of the visitors that reach the initial page do not go on to a discussion (because they do not see any interesting one, for instance), and because people may also reload, but it is still a good deal over the 300 or so that published their opinions non-anonymously.
12 There is one more option: instead of replying in public, you can send an e-mail message to the author. But then the whole point of sharing your views with the community is lost.
13 #2254080, an anonymous participant reacts strongly to the idea of using one-letter variables. We shall study this message in more detail in the chapter about *coding styles*, which shall include an explanation of what one-letter variables are. Suffice to know here that they are a small technical programming detail.
14 There is another thing that most slashdotters have in common, namely a keen interest for open-source projects. This does not mean that all slashdotters contribute to the open-source efforts but it does mean that most of them sympathise with that movement. Hence, in Slashdot, Windows is bad and Linux is good; in general, anything that comes from Microsoft is bad: operative systems, applications and programming tools.
15 It is unclear, for instance, whether such a concern will result in cheaper, more useful, better selling or, in a general sense, more successful (in a public sense) programs. In fact, it is not sure that such a concern will result in more beautiful software.
16 It is important to clarify that what I know about programmers deals mostly (if not exclusively) with those from Western societies. I know precious little about programmers in other cultures, and not because

they would be less important (for instance Japan, China and India have extremely powerful software industries). I have the feeling that one might find among them similar phenomena as the ones I present here, but this is really just a hunch. Better informed studies are certainly welcome.
17 There are huge systems and small applications; there are programs for chips (in cars, robots, airplanes, microwave ovens, etc.) and programs for proper computers (and there are many kinds of computers, from programmable calculators to "big blues"), programs that are wired to the hardware and programs that can be maintained, and so on and so forth.
18 Programmers usually use more specific labels: phreakers, warez doodz, virus writer, sneakers, samurais and dark-side hackers.
19 The coding part of programming is often considered secondary to design, but more about this in the first paragraphs of the chapter on *coding styles*.

III
Programming

I was myself a programmer and in this chapter I propose a simplified version of one of the projects in which I was involved, with the purpose of introducing the subject of this thesis, the private aspects of programming.

I cannot just suppose that everyone knows what programming involves, but on the other hand, neither can I explain all its technical details. So I thought the best thing would be to give an idealised example, which does not really transmit all the details but that is near enough to reality to give a context to the expression 'private aspects of programming.'

What about an example from my own brief career as a programmer? A potential customer came to our door and wanted an application to be done for him. He needed a program that could manage the data about his past, present and future sales (including lists over customers, kind of products, suppliers and so on). At that time (middle of the nineties) we, my associate and I, were very much into Delphi, a programming environment developed and sold by Borland. It included database and gui (graphic user interface, pronounced 'gooee') tools and a programming language that was an object oriented variation of Pascal. We liked the environment and had already written a few programs in it. We proposed to our customer that we use Delphi but he was not so interested since his company had already bought licenses for Microsoft's Access, another database programming environment, about which we also had some knowledge. So, against our wishes, we had to program the application in Access. Now the interesting thing here is why we preferred Delphi to Access.

Access was not as versatile as Delphi (I have not followed the development so I do not know what a comparison may yield nowadays), but it also served the purpose well. So we did not prefer Delphi because the problem could be better solved with it, in fact, it may have taken more time, since Access had more ready-made elements in it. It was a rather down-to-earth application and it did not require advanced programming tools. The application written in Delphi would neither be noticeable faster nor more efficient than in Access, the database was too small to allow any real difference. The network across which the application had to work was a simple and straightforward Windows LAN, five computers and a few cables. So there was nothing to be gained by an intelligent use of the network, something better tried on Delphi than on Access. Neither did we prefer Delphi for economic reasons, since the customer offered us one of his Access licenses. So, why did we prefer Delphi?

Simply because it is a finer tool than Access. The computational problem might be more efficiently solved (cost less and require less time) using Access but we were not just solving a computational problem, we were programming. And to program includes other things, such as writing code that you are proud of. And we did not feel Access was subtle enough to let us write what we wanted. Not that we were going to show the program to the whole world, but still. A matter of private pride, one could say.

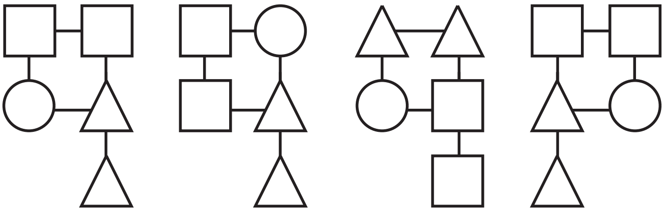
We finally accepted Access, but some of the programmers that will appear later may have totally refused to work with it, regardless of the alternative, for a single simple reason: it's Microsoft. On the other hand, others may have refused to work with Delphi: it's Pascal. Working with Pascal, or with Microsoft, is considered by some programmers as shameful, a proof of lack of taste. It's just not done, and if unhappy circumstances force one to, it is a sign of good manners to clarify one's opinions:

Java is inefficient by **Procrasti** (#2253348)
Even Java works squarely against the goal of "efficient". Give me C++ any day.
I've done projects in C, VB (*im not proud*), C++ (yep MFC et al, 5 years) and Java (1.5 years now), and I question the statement that java isn't efficient. [...]

The emphasis is mine, and VB stands for Visual Basic, another of Microsoft's programming tools. This message is part of a Slashdot thread that dealt with the difficult subject of comparing Java to C++. Some programmers favoured Java and some others C++, and the discussion did not seem to change anyone's preferences. Refusing to work with Microsoft products, or being ashamed of doing it, is however not only a question of aesthetic preferences. It may also be a political statement, and it is not always easy to know where to draw the line. Some programmers consider Microsoft evil and rude[20], and the aesthetic dislike of their products may very well be influenced by its monopoly. The purpose of this thesis is not to delineate the causal relationships between the political and the aesthetic, but to discuss the effects that private aspects of the relationship between the programmer and the program, so I shall leave that discussion here. There will be an opportunity to return to Microsoft's poor reputation in certain programming circles, and more specifically what using Visual Basic may do to one's image as a programmer.

All right, so we would have preferred to work with Delphi but had to use Access. The aesthetic aspects of programming are of course not limited to liking and disliking programming environments, or preferring one programming language to another. Let me continue with the idealised example.

Once it was clear that we were going to program in Access we started to design the application. This was not a very complex program. In fact, it was something quite elementary with a time frame (including all the usual misunderstandings with the customers, the operative system breakdowns and sorting out the network bugs) of about two to three months and with a cost of less than €4000. Designing such an application requires familiarity with databases and programming languages, i.e. some technical knowledge. But the process of design should not be understood mainly as the *application* of this knowledge, as if designing a program was a matter of deduction. Instead, it requires the programmer to *create* something. And I do not mean to imply anything glamorous in this 'create', it only represents the unspectacular notion that where there was nothing there has to be a program[21], and that there are decisions that cannot be deduced but that must nevertheless be made. Things like "what sort of objects do we need, how are the connected, how does one access them, what operations do we need to carry out on the data, etc." These decisions are interconnected, of course, and choosing the properties of an object, for instance, will influence the mode of access to it. In such a small application the relationships are more or less easy to sort out but they still have to be designed. Needless to say, there are no formulas to apply and one has to choose among the different alternatives without the help of a scientific model that calculates the best solution. The following figure explains the situation:



FIGURE, All solutions work (represented by all figures having the same connections), all are feasible, all cost more or less the same, the customer won't see the difference. Which one do we choose?

To program involves therefore not only applying one's technical knowledge but also to making decisions that cannot be based on scientific laws: one loop or two loops? one or two objects? these properties or those? and so on. There is nothing in the way of a science of programming with a formula to tell which solution is most appropriate. The Computing Science has no answers to these kind of questions.

The grounds upon which programmers make these technical decisions are varied, and it is not easy to distinguish between them. Programmers may base their decisions on economic rationality, or on aesthetic preferences, or on tradition, or even on beliefs (about the users' needs, about the economical, functional or political consequences of certain technical strategies, about the longevity of the application, etc) and it may prove impossible to distinguish one ground from another in real life situations. Real life is much messier than what the result of sharp analysis leads us to believe. This thesis deals in fact with the pre-analytical mess that one encounters when reading what programmers say about programming and when trying to explain some programming behaviour. My analysis tries to avoid the reductive mistake made when considering the programming project (and programming in general) as a series of objective choices, and hence, an activity that progresses according to a coherent course.

Programmers are perhaps partly to blame for this mistake since they sometimes recount their experiences following just one such coherent course. For instance, they may explain (legitimise?) their decisions with arguments of economical rationality: trying to minimise costs. However, this is not as straightforward an approach as it may seem at first. The concept of 'minimising costs' is, on the contrary, a rather thorny one, since it is not clear what makes a program cheaper. Is it cheaper if it is written faster? or if it can be used for a long time? or if it contains fewer bugs (and requires less maintenance)? or if one does not need to buy licenses to develop it? It is seldom possible to find final answers to these questions, for reasons that could well be explained as bounded rationality (Simon 1997): programmers do not have access to all the information needed. In fact, according to Wittgenstein, it may be impossible to gather all the information needed, not due to errors in the search but to the intrinsic properties of the world, more specifically of language. In other words, it is not our laziness, it is not that we do a deficient job of finding all the information needed to calculate the cheapest option; no, in fact, it is impossible to obtain all that information[22]. But leaving Wittgenstein aside for the moment, and going back to the various aspects of programming, we shall see that when programmers make decisions with the intention of minimising costs, their reasoning is not based on comprehensive calculations but on assumptions that originate in experience, hear-say, beliefs and other things that cannot be labelled Computing *Science*. Hence, even when programmers argue in economic terms, the private aspects of programming (their beliefs and experience) have effects on their applications.

In some cases the economic discourse is not even raised, perhaps due to a clear insight of its flaws, perhaps due to lack of interest. Another usual way to reason about the technical decisions (about programming, indeed) is along the lines of functionality. The functional discourse centres on the usefulness of software: programmers will explain that they program with the purpose of creating useful applications. Functional rationality is however also flawed, and in a similar way to economic rationality: programmers cannot possibly know exactly how their technical decisions will effect the usefulness of the program. This does, naturally, not prevent them from making decisions according to what *they think* will be more useful to the customer. Another difficulty with this line of argument about technical decisions is that often the alternatives present to the programmer make no difference at all to the customer, since their problem can be solved satisfactorily in a number of ways. How can then a programmer decide based upon the needs of the user? Yet another difficulty is that programmers may, for a number of reasons, misunderstand the customers' needs totally: they may be uninterested, or tired, they may lack the capacity to place themselves in the position of the users, or the users the capacity to explain their needs, etc.

In some cases the discourse does not include any advanced legitimising manoeuvres, programmers will simply decide they want to use Delphi because that is what they always have done, or because the company requires them to do so. In other cases, there will be direct reference to personal preferences ("I do not like to work with Visual Basic"), to political convictions ("never Microsoft") or even to aesthetic opinions ("Perl is ugly").

This thesis is interested in the mentioned analytical mess that appears when studying what programmers say about that creative activity that is to program. They explain how they program, and what they think about programming and programs, with the help of all kinds of notions: economics, functionality, aesthetics, convictions, all of them relying upon each other ("application X is more beautiful because it is more efficient" kind of statements)… it's turtles all the way round. But the mess does not appear clearly unless one reads the programmers closely, or unless one is lucky enough to hear them discussing about programming itself, as opposed to the uses of a given application or some technical details in an operative system. I was lucky, I found two discussions, presented in the chapter on method, in which programmers exchanged opinions about the aesthetics of software, giving me an inroad into the private aspects of programming.

So I suggest that we prepare our visit to the disordered world of the private aspects of programming by presenting an analysis of the concept of instrumental goodness, i.e. the notion by which we designate a good program.

20 *Evil and rude* according to the Jargon File (also known as The New Hacker's Dictionary): "Both evil and rude, but with the additional connotation that the rudeness was due to malice rather than incompetence. Thus, for example: Microsoft's Windows NT is evil because it's a competent implementation of a bad design; it's rude because it's gratuitously incompatible with Unix in places where compatibility would have been as easy and effective to do; but it's evil and rude because the incompatibilities are apparently there not to fix design bugs in Unix but rather to lock hapless customers and developers into the Microsoft way. Hackish evil and rude is close to the mainstream sense of 'evil'" (Raymond 2003)

21 I say unglamorous because one is likely to associate 'creative' (particularly in the proximity of 'design') with exciting themes, such as hip interior architecture firms, Italian cars, expensive and innovative gadgets or ground-breaking art. Programming can be as exciting as any of these but that is not the point. The point is that one creates even when writing the most boring application, that indeed, every artefact has been created by someone, it has gone from being a fantasy to becoming reality. An observation that I owe to Gustafsson's reading of Vigotskij's *Fantasi och Kreation i Barndomen* (Gustafsson 1994) (Vigotskij 1995)

22 Ludvig Wittgenstein – perhaps the most influential philosopher of the 20th century, and definitely the most influential philosopher for this thesis – dedicated a large part of his work to the investigation of language. Trying a summary of his work in a footnote is almost stupid but I may get away with it if I concentrate on just one of his points, using one of his analogies: words are not like bricks, perfectly delimited blocks that form a wall (language) by being placed one by the side of the other in an orderly manner; language is more like a rope: its strength is the result of threads going into one another in a vague and rather chaotic way. Hence, information based on language, as opposed to information based on mathematics, is obtained from a certain number of words whose meaning go into each other and whose boundaries are vague. Expressions like 'find all the information needed to decide which alternative is cheaper' sound grammatically correct, but they are, in fact, confusing: the very idea of there ever being enough information to be able to decide which alternative is cheaper is based on an incorrect assumption about how language is constituted and what can be done with words. "All the information needed to calculate which alternative is cheaper" is simply an absurd statement, the information we may obtain about the world cannot be organised in such a way as to allow us to carry out such a calculation (Wittgenstein 1969, 1997)

IV
Instrumental, Semi-Instrumental and Intrinsic Goodness

Messages in the examined discussions among programmers often feature the adjective ‘good’. The first analytical task is to sort out the different kinds of goodness that programmers refer to. This chapter is dedicated to this task, and to the introduction of other concepts that will be used later on.

In the previous chapter I proposed a simple example of a programming project, with the aim of introducing the notion of 'private aspects.' This chapter has also an introductory function; it presents a number of secondary concepts that will be used later on. These concepts are the result of my analysis (hence "secondary") and are not used, or not in the same way, by the programmers.

As you can notice, I present them *before* the empirical descriptions, a decision that was not made lightly. Doing it so implies, on the negative side, that the readers must approach them without the help of the descriptions; however, on the positive side, they will be prepared for them. My intention is to create some basis from which to interpret what comes next. There are not final solutions to this situation, but I hope that the argument is brought forward more clearly this way.

The main contribution of this chapter is the introduction of three different kinds of goodness: instrumental, semi-instrumental and intrinsic. Since we are going to study the private aspects of programming, and these aspects are manifested – partly – in the existence and importance of intrinsic goodness, we really must give this concept some attention. This is best done, I believe, by comparing it with its opposite: instrumental goodness, which is one of the public qualities of software. These two are related to each other in complex ways, which can be clearly seen in the ambiguity of some of the programmers' commentaries.

If this had been a world properly ordered according to analytical principles, we should have only needed these two kinds of goodness: the instrumental and the intrinsic, or perhaps only the latter one. It is indeed possible to imagine that programmers speak either about the beauty of code or about its utility, that is about its private or its public aspects. The problem is that, as mentioned earlier, the programmers' comments are sometimes ambiguous and it is difficult to know exactly whether they refer to beauty or to utility. Furthermore, sometimes the programmers are not in a position from which to evaluate the utility of a program, since they are not the final users. What happens then, from an analytical perspective, when they say that a program is useful, even if they cannot know if it is or not?

This class of affirmations give rise to what will be called semi-instrumental goodness (it could also have been called semi-intrinsic), and the mechanisms behind it play an important role in the creation and maintenance of the private aspects of programming. But let us start, as usual, from the beginning.

INTRINSIC GOODNESS

MacIntyre (MacIntyre 1985) tells the story of the young girl who was tricked to play chess by an older relative. He promised her candies for every match she won, and made sure he lost regularly. At this stage, it is assumable that the girl played chess because of what victory reported her, not because she enjoyed it particularly. With time and practice, however, MacIntyre tells us how she started to enjoy playing, and even ceased cheating (to cheat was fine as long as winning was the essential). She had reached a position where she was capable of appreciating the intrinsic qualities of chess, she played chess for the pleasure of it, as opposed to for the promise of prizes.

A similar case can be made for programmers, but it is not based on the idea that they would program just for the pleasure of it. Some of them clearly do (consider the magnitude of the open source movement, or read Linus Thorvalds own version of the making of Linux (Torvalds and Diamond 2001)), but their efforts are only the most radical manifestation of the private aspects of programming. The basic element of those private aspects is instead the appreciation of the intrinsic qualities of code, and the relationship that such an appreciation creates between the programmer and her code.

How can you know that someone feels an affinity towards something? Only by looking at what they do and what they say (… and by assuming that they do not lie, of course). Programmers, for instance (and this is the phenomena presented in the following chapters), speak about their code, they engage in disputes about what is the best programming language, they write their programs according to programming styles, they defend their personal preferences, etc. Their code, and this is what the empirical material will hopefully show, speaks of them: of their preferences, their skills and their assumptions. This is the substance of their relationship towards code, but, strictly speaking, it does not imply that there exists (or that they care for) an intrinsic goodness of code.

A concern for the intrinsic goodness of an object is a concern for the value that this object has in itself, regardless of its use. Now, programmers could very well identify with their creations and nevertheless not worry about anything else but their utility. It could be that their only concern was for what the utility of code said about them. Is it useful? and cheap? and easy to use? Such a concern would only have to do with the instrumental (public) aspects of programs, leaving little possibility for private aspects. Programmers and the general public (users, managers, investors, etc.) would have exactly the same perspective on code, and this thesis would not exist.

This thesis does, however, exist, and the reason is that programmers do care for other details in their program apart from their utility, cost and ease of use (to ease the reading, from now on I shall only say usefulness). This does not mean that they do not care about its usefulness, only that they care about some other aspects as well, and that these aspects are not necessarily related to it. These aspects are, in themselves, of no interest for others than the programmers themselves; in fact, they are often totally invisible to non-programmers. This is why I speak about *private* aspects of programming, and this is why it is fruitful to think in terms of intrinsic goodness: we are interested in the fact that programmers care (also) about code itself, regardless of its utility.

Perhaps a pertinent question here is: "how much do they care?" Well, they do not all care as much, they do not all care in the same way, and they do not all formulate their care in the same way. Some care a lot, and some care very little, but drawing this landscape is what this thesis is about, so I suggest we wait with all this. Let us simply state that intrinsic goodness is a measure of the quality of code in itself, regardless of its usefulness, and that this is the basis of this thesis.

INSTRUMENTAL GOODNESS

Instrumental goodness is a concept that I have borrowed from Georg Henrik Von Wright's *The Varieties of Goodness* (Wright 1972). This work of his is a study of the meanings – uses – of 'good', its purpose being to serve as a partial "prolegomena to ethics" (:2), not aesthetics. This does not mean that instrumental goodness has a moral sense, only that Von Wright considers that the "so-called *moral* sense of 'good' is a derivative or secondary sense, which must be explained in the terms of non-moral uses of the word" (:1, italics in original).

Von Wright says that "instrumental goodness is mainly attributed to implements, instruments, and tools – such as knives, watches, cars, etc." (:19) and that "[t]o attribute instrumental goodness to some thing is *primarily* to say of this thing that *it serves some purpose well*.

An attribution of instrumental goodness *of its kind* to some thing presupposes that there exists some purpose which is, as I shall say, *essentially associated* with the kind and which this thing is thought to serve well" (:20, italics in original). We say that a hammer *is good as a hammer*, meaning that it serves well the purpose essentially associated with hammers (to drive in nails).

Now, since programs are tools, i.e. they are used to achieve some ends, it is perfectly sensible to speak in terms of a 'good' (in the instrumental sense) program. This goodness, from some different perspectives, is what the public discourse about software deals with. For some, a program is good if it really helps them carry out their jobs, for others if it creates succulent revenues, for others if it helps them open new markets, and so on.

But we are not interested in the views that non-programmers hold of programs, this is a thesis about the private relationship between programmers and their creations, and we shall only deal with other interests marginally. Now programmers also care about the instrumental qualities of programs (including those they write), they are well aware that they play different roles (tools, products, services). Only in very rare occasions will we find programmers totally ignoring the instrumental sides of a given program: a program is practically never written without a goal, just for the pleasure of putting commands together. Programs, so to speak, always have a mission, however small it may be.

So it would seem that we are drifting away in our analysis of the instrumental and intrinsic goodness of software. These two are apparently easy to distinguish, and the reader might be wondering why this issue was brought up at all. Ok, you could say, a program has instrumental qualities, but this thesis is about the private aspects of programming and they both seem unrelated to each other.

Obviously, there is more to say. The world seldom complies with the analysis, there are always problems. In this case, one of the main problems emerges with the notion of the 'essential purpose associated with a tool'. What is the essential purpose associated with a program? And is it possible at all to assess how well the program serves this purpose?

SEMI-INSTRUMENTAL GOODNESS

The purpose essentially associated with a program may be a difficult thing to pin down. In some cases, it is certainly very easy (consider for instance the small application that shows the current time on a corner on your desktop), but what about somewhat more complex applications, such as word-processors? What is the purpose essentially associated with them? To write letters, or to write books? or scientific articles, or notes, or faxes, or bulletins…? And how do we decide that one processor is a better tool than another? The only way to do that is to simultaneously describe in what sense (for what purpose) it is better, for instance by saying the word processor X is better *if you need it to be compatible with the Internet*.

Furthermore, programs are a particular kind of instruments, since they run on computers (and must therefore follow hardware innovations). Hence, someone may think that a good word processor, that serves its purpose well, is one that will continue to exist when Operating Systems change. Or this same person may instead reason that, since there are always things to improve in a program, a good word-processor is an application that is constantly – and easily – maintained (bugs are fixed, new functionalities included, old functionalities discarded, etc.).

Now, word-processors are by no means the most complex of programs, consider for instance the difficulty of deciding whether an Operative System is (instrumentally) good or, even worse, whether systems for airline ticket booking or mobile telephone roaming serve well the purpose essentially associated with them.

For instance, specifying the essential purpose of a roaming system may sound like an easy task, perhaps it is simply to keep track of the location of all the mobile telephones in the network. But the expression “to keep track” is deceivingly simple, something that becomes clear when you compare two systems in order to decide what it is that makes one better than the other: shorter delays? Fewer lost mobiles? Freedom of choice? Cost of initial investment? Cost of maintenance? Upgradeability? Customer satisfaction?… Which of all these things is more essentially related to keeping track of mobile phones? Or, more to the point, which one of all these is the most important to the user? After all, it is the user who defines the essential purpose associated with a tool.

So, defining the essential purpose of a program is difficult, but what does it have to do with the private aspects of programming?

It has one very important thing to do with it, because it happens that programmers hide private aspects of programming behind a seemingly instrumental discourse, and this phenomenon is most visible in the cases where the essential purpose of a program is difficult to pin down. In order to give a good example, I need to explain a programming concept.

The concept of readability will be treated in detail in the next chapter, for our present purposes it suffices to know that the code of a program can be more or less difficult to read. With this, programmers mean that the code can be more or less difficult to interpret, in other words, that it is more or less difficult to see the structure of the program, what each component is used for, why, in what order, and so on. Readable code is easy to understand (and to fix and modify, if need be); unreadable code is a pain:

Re:nice, but welcome back to the real world by **Myopic** (#2253074)
[…] I have NEVER turned in to my boss anything but well-documented, well-commented, readable code. I don't do this out of respect for my users; frankly, I know how to use the software and if they don't they can read my docs and try to figure it out. No, I do it for the other schmucks like me. At some point, my boss will probably tell his next lackey to add some little feature to one of my modules, as he's asked me to do with some older programmer's works. And it's DAMNED IMPOSSIBLE to wrap my head around code which is all mixed up. I comment for other programmers. People who might need to sink their hands into my code.

Myopic is answering to other participants who said that the idea of writing beautiful code sounds fine but that real world conditions do not allow for such niceties. Some others, like *Myopic*, reacted by explaining why it is important to write readable code (in this thread, beautiful and readable were interchanged freely, we shall see more about this issue in the following chapter). *Myopic*, and some others, claimed that you should write readable code out of kindness. His/her motto is the classic 'you should do to others as you would like others to do to you.' This kind ethical consideration appears here and there in discussions about readable code, and it forms part of the private aspects of programming: this is one way of relating to your code, this is one thing you want your code to say (that you care for other schmucks like yourself).

But right now we are more interested in another kind of reasoning around the idea of readable code. *Myopic's* comment continues:

Paying me now to write comments and format things well is worth it for the added speed with which the software will be maintained in the future. So for me, and I'm sure most of the code jockeys on Slashdot,

> the "real world" is one where software is written, THEN MAINTAINED. Beauty is part of maintanence.

Here s/he suggests that it makes economic sense to take the time required to write readable code, and this is definitely a reason that pertains to the public discourse. It is fitting that we find both kinds of reasoning (the ethical and the economical, the private and the public) in the same comment: this is what opinions usually look like, a bit of one reasoning and a bit of another reasoning. This is not due to any analytical shortcoming in programmers but rather to the nature of everyday language.

At any rate, the interesting point in *Myopic's* latter excerpt is that s/he makes an economic statement that is based on an internal quality of software. What do I mean by this? That Myopic connects the internal (private) quality of readability with the external (public) quality of cost.

internal characteristics of the code	implies ▶	external characteristic
readability	implies ▶	less cost, easier maintenance

FIGURE, External qualities are implied from internal form

Now, is this connection necessary? No, it is not. There is no casual law, not even a statistic law that would connect these two. To start with, it is difficult to assign the cost of a software development project to particular posts: is it really more expensive to write readable code? or cheaper to maintain readable code? The development and maintenance of software (at least of projects of some size) are hugely complex enterprises, and my immediate impression is that it is impossible to answer those two questions in a general sense. In fact, we still do not know whether it is more or less expensive to have larger teams of programmers, or a particular kind of documentation, or a particular kind of organisation… all these are characteristics of a project that, very much like the readability of the code, are only very loosely connected to the costs.

Now, in the case of readability, there is an added difficulty when trying to connect it with the cost of the project, namely the vagueness of the concept. As I said, this concept will be studied closely in the next chapter, but I can advance one of the results: readability is not a homogeneous concept, i.e., programmers do not agree on what it is that makes code more readable. Some say, for instance, that a lot of comments (we shall see what those are) make it so, others that a lot of comments clutter the code and make it more difficult to read. Readability is, in other words, a subjective quality of code.

It may also be argued that by 'readable' we mean that a program is understood by a majority of programmers, that the concept has a statistical meaning. What we face here are two different uses of the same word. A number of scientists, with a particular approach to software development, use it in the statistical sense, putting aside their own, or any other individual, opinion. 'Readability' is, for them, a characteristic of software that can be measured statistically, code A is x% readable, so to speak. This can be measured by having a number of programmers read the code and answer questions about it. The validity of these measures is an unresolved question, but that is a problem of the structure of the experiments, not of their concept of readability.

On the other hand, programmers, without laboratory data to draw from, can only rely on their own judgement. When *fishbowl*, discussing a rather obscure piece of Perl code offered by *quartz*, tells the latter that the presumption that your code is somehow ‘unreadable’ bothers me... there’s nothing in this example that should be a problem for even a beginning perl coder, in my opinion. You’ve used a common perl idiom in a very efficient, clear, understandable way[23]

s/he is not using readability in the statistical sense. This is his/her own subjective opinion of the readability of *quartz*’ example. In fact, other participants found it difficult to understand (an anonymous participant replied: “I'm a moderate level perl coder (one step above beginning), and I have no idea what map or ucfirst means...”[24]).

The version of readability that I am interested in is the one used by programmers – I am, after all, studying *their* experience of programming –, so I must consider it as a subjective trait of software. For our purposes, hence, reading code is not a statistical phenomenon but an individual one.

These considerations make questions like “is it really more expensive to write readable code? or cheaper to maintain readable code?” quite impossible to answer in general terms. Myopic’s statement must hence be treated carefully.

At the same time, I do not think *Myopic* is thoughtlessly stating things, s/he really believes this connection (readable - less costly) holds. This is his/her belief. We shall not go into the motives you may have for holding such a belief but it probably has to do with your experience, what you have learnt at school, what you hear from other (admired) colleagues, etc. *Myopic* is not making things up. Furthermore, I am by no means saying that the opposite holds (readable - more costly), only that there are no general rules that apply.

Someone might think that I am somehow belittling *Myopic’s* programming skills, when I suggest that his/her statements are not universally valid. This would be a mistaken judgement, based on the following train of assumptions and conclusions: that programming can either be done correctly or incorrectly, that it is a matter of calculation, and that *Myopic* is not calculating but feeling his/her way forward. Now, this whole thesis is an argument against this idea, programming is not only about calculating optimal solutions, it is also, in a deep sense, a matter of personal decisions. This does not mean that some programmers are not better than others, simply that none of them limit themselves to calculations. To follow personal beliefs when programming, as *Myopic* does, is not a shortcoming but the only possible way to program.

Well, this last statement is not quite correct. There is another option: to program, according to your personal preferences, and not make any connections between them and the public qualities of software. *Myopic* could just have said that one should write readable code out of kindness to other programmers, or that s/he writes readable code because s/he likes to. Now, presenting one's preferences exclusively from the private perspective is not easy. Mainly because it is not legitimate but also because this is not the way programming is generally taught and it is not the way programming is discussed.

At any rate, there are beliefs that connect the internal qualities of software with the external ones. I have called them 'instrumental beliefs', because they are used to make statements about the instrumental properties of software, when such statements cannot, from a strict perspective, be made. The role of instrumental beliefs can be illustrated with the following figure:

$$\begin{tikzcd}
\text{external quality of program} \\
\text{instrumental belief} \arrow[u] \arrow[d] \\
\text{internal characteristic of code}
\end{tikzcd}$$

FIGURE, Instrumental beliefs connect internal form with external qualities

They are often used unconsciously, so to speak, and in rather vague forms. For instance, few programmers will go as far as saying, explicitly, that readable code yields better tools. Instead, they will say that readable code yields better software, without defining what kind of goodness they are referring to. In fact, they will more often speak in personal terms, explaining how they think one should program and loosely claiming that programming so yields better software. *Myopic's* is, once again, a good example: "Paying me now to write comments and format things well is worth it for the added speed with which the software will be maintained in the future." S/he invites the reader to presume that software that is more speedily maintained is better software, but does not say so explicitly.

I have therefore decided to include a third kind of goodness, the vague one that programmers refer to when they express instrumental beliefs. This is the semi-instrumental goodness, which could also have been called semi-intrinsic because it is unclear which one *Myopic* means. The analytical value, and elegance, of such a concept is questionable, but I think it points to an important aspect of the personal relationship between programmers and their creations, namely the unclear border between the public and the private discourse. When programmers discuss programming, both kinds of reasoning appear often close to each other: beauty, for instance, is mixed with maintenance, which is directly connected to costs and utility. The private aspects of programming are manifested very clearly in particular phenomena (such as aesthetic discussions, harsh disputes and other details that we shall be studying soon) but also, albeit less clearly, in their comments about what it is that makes software better.

In this chapter I have showed how programmers use the concept of readability and how this use points to the existence of a hybrid kind of goodness, not strictly instrumental but neither properly intrinsic. Perhaps you could say that it is a mixture of private preferences with instrumental legitimacy… but, as I said, the concept of legitimacy will be dealt with later on.

Now, are there other concepts, apart from readability, that can be used in a similar way? Structure and robustness are both of the same category as readability, they refer to the properties of code, but can be used to indicate a kind of goodness that is vaguely related to the instrumental qualities of the program. These two properties will later on be briefly presented in the chapter on *aesthetic ideals*, readability could have accompanied them, but is instead treated more in detail in the chapter on *coding styles*.

Efficiency is a bit special since it may refer to different things: at the programming level, it has to do with optimum use of memory, or lines of code, or disk-space or something like that. This is a quality of the code itself. But it also points to an external quality, namely that of solving the users' problems in an efficient manner. However, there is no necessary connection between an efficient use of the memory and solving the users' needs. Naturally, these two conditions are not opposed, and in some cases they may actually be directly related.

Functionality is a characteristic of software rather difficult to define, but paradoxically, it is also one of the most important. It is indeed so important, and its vagueness so representative of the private aspects of programming, that I have dedicated a whole chapter to it. The chapter *beauty and functionality* is a study of the relationship between the most obvious private aspect of code and the most obvious public aspect of applications.

I believe it was important to present these concepts to provide analytical support before plunging into the richness of the empirical material. The two following chapters are much closer to programming, both presenting different programming alternatives and at the root of the private aspects of programming. Without the possibility of choosing between different alternatives (and still obtaining the same result), there would have been no personal relationship to code, only calculating and optimising. Let us turn now to the most concrete of options: coding styles.

23 Slashdot message #2253306, a more detailed explanation of this exchange can be seen in the chapter on *coding styles*.
24 Slashdot message #2253979

V
Coding Styles

Private aspects of programming are not limited to what could be called 'aesthetic' aspects, but these provide a good concrete ground from which to approach them. They give us something to get hold of, so to speak. However, as programming consists in the manipulation of abstract structures, attempts at giving examples of beautiful software to non-programmers usually end up in long technical explanations. In order to avoid this, but still present a concrete example of the aesthetic possibilities available to programmers, I propose to examine one of the most straightforward parts of programming: coding. Coding is considered by some programmers as a peripheral concern but, on the other hand, it results in something tangible (code), allowing thus for good illustrations. Besides, however secondary it may considered by some programmers, we shall see that coding aspects are important enough to spark off heated disputes among them.

For some readers, this may be the first contact with the field of software aesthetics, well, perhaps with code at all, and there are a few clarifications that need to be made. I shall start by explaining what 'coding' is and why this chapter is called *coding* styles instead of *programming* styles.

'Coding' is a part of the activity of programming, perhaps I should say, a phase of it. The general idea is that, once the users have explained what they need, the whole thing starts with the creation of a guiding document, called 'technical specifications', which is a description of the technical characteristics of the program. Directed by this document, a programmer will produce a design: a more or less detailed description of the structure and the elements that will be needed to construct the program. Exactly what this structure must include and what kind of elements (building blocks) are available to programmers would take too long to explain, suffice to say that the design functions similarly to the blueprint of an engine, in which the different elements and the relation between them are drawn. Once the design has been – hopefully – revised (so that it 'holds', so that it mirrors the technical specifications), the programmer starts to code, which is nothing else than typing the necessary commands so as to construct the design in a form that a computer (or microprocessor) understands.

Programmers sometimes give designing a higher status than coding, reasoning that it requires more knowledge to be able to solve the problem (design) than to translate the solution into commands (coding). In fact, occasionally, project organisations differentiate between designers and coders, the former ones generally being better paid. Also, the word 'coders' can be used pejoratively, meaning poor programmers. However, it is not always easy to distinguish between the three phases I have mentioned so far: specifications requirement, design and coding, particularly when they are all carried out by the same programmer. They blend into each other, often quite deeply, making it impossible to draw any clear lines. Hence, often 'to code' means 'to program', and 'good code' means 'good software'. But as much as it is impossible to draw clear lines, the results of the three phases are three different documents: technical specifications, design and code; and these are almost impossible to confuse. Of all the three, the coding is the one best suited for illustrating the aesthetic possibilities offered to programmers, apart from being the one that is easier for them to exchange (and for the researcher to find examples of).

Now, why have I decided to start with a presentation of coding styles, instead of a presentation of programming styles in general? This has to do with the mentioned availability of documents. Since some, perhaps many, of my readers have never seen neither code nor design, it is important to thicken the description as much as possible and show examples. And when it comes to this, code is a much better alternative than design documents, not to speak of technical specifications: code snippets are freely available all over the internet, they are much easier to reproduce than design and they are often used by programmers to illustrate what they mean. Also, it is easier to point out the fineness of coding details than the elegance of a design without going into much technical overhead. Code is simply much more immediate. In the next chapter, we shall have occasion to discuss aesthetic ideals in programming at large, but I think it is better to introduce the subject with code examples. I hope that in the course of the presentation of coding styles the reader will get an insight into the possibilities of aesthetic expression available to programmers. After that, it should be easier to understand the more general aesthetic ideals.

Before we start for real, there is something to be said about the comprehensiveness of this presentation of coding styles. In programming, as in the majority of disciplines of human activity, there are different, sometimes opposed, notions of what it is that makes something beautiful. A complete classification of all these ideals for the case of programming coding styles would be a nice thing to present, but I cannot do that. The main problem is not that there are so many programmers scattered all over the world, not even that they lack a formal organising body; the real problem is that beauty, in programming, is a private affair, something kept among programmers and absent from the official discourse: programmers are not supposed to spend their time creating beauty, they are supposed to create programs that work. Nowhere in a technical specification, the formal description of the program to the customer, will one find the word 'beautiful'; strictly put, no-one is ready to pay for beautiful software and there is nothing like a body of code-critics, as there is in the realm of art. As a result, the idea of software aesthetics is mainly discussed in small groups, and generally at the local level. Given the amount of programmers, and the subsequent amount of local levels to which I have no access, it would be unwise to claim that the classification presented here is anywhere near comprehensive. However, the task at hand here is not to achieve final classifications (neither of coding styles nor of aesthetic ideals in programming) but to give an ordered presentation of some manifestations of aesthetic judgements among programmers. I do not seek complete classifications but a discussion of the programmers' personal experience of programming.

As I mentioned before, the distinction between coding and the rest of the programming activity is not always possible. The name ‘coding’ is sometimes used to denote the translation of ready software designs into commands, but there is much more going on than just translation when a programmer writes code. At any rate, coding is not a mechanical activity by which a complete design is translated into lines of code; regardless of the detail with which the designer has specified the program, the are numerous alternatives available to the ‘coder’. The same design can be translated into different codes that are identical from the perspective of the processor, as the figure illustrates.

$$\text{design}\left\{\begin{matrix}\text{code 1}\\ \text{code 2}\\ | \\ \text{code n}\end{matrix}\right\}\text{processor actions}$$

FIGURE, One design, one function, many codes

The good thing about studying coding more in detail is that, as I said, *the result* of coding is very concrete: lines of code. This concreteness makes it easy to give examples that explain the alternatives and the styles. For instance, the following two codes make the processor carry out *exactly* the same actions:

```
var
  numberOfElements, i : Integer;
  list array [1..4] of String;
  aux: String;
  listIsOrdered : Boolean;
begin
  // we initialise the variables
  numberOfElements := 4;
  list[1]:="G"; list[2]:="Dn";
  list[3]:="S"; list[4]:="Dv";
  // here starts the ordering
  repeat
    for i:=1 to numberOfElements-1 do
      begin
        istIsOrdered:=true;
        if list[i]>list[i+1] then
          begin
            // theyre not ordered, we change them
            aux:=list[i];
            list[i]:=list[i+1];
            list[i+1]:=list[i];
            listIsOrdered:=false;
              // we have changed something
              // in the list, we are not sure
              // the list is ordered
          end;
      end;
  until listIsOrdered;
end.
```

CODE, Alternative 1: Typical layout

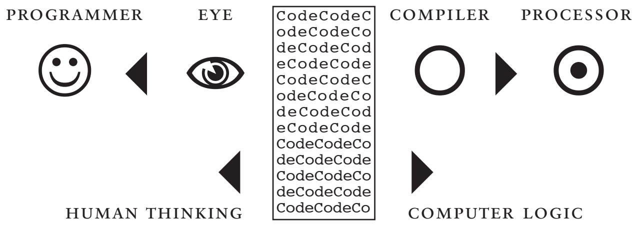
```
                 var
wwfgthe, kjlu : Integer;
     lksswer array [1..4] of String;
               klfft: String;
oggtlei: Boolean;

begin
// well she’s walking through the clouds
// with a circus mind
// that’s running wild
// butterflies and zebras
      wwfgthe:= 4;
            lksswer [1]:=”G”;
     lksswer [2]:=”Dn”;
lksswer [3]:=”S”;
               lksswer [4]:=”Dv”;
// and moonbeams and fairy tales
// that’s all she ever thinks about
// running with the wind


              repeat
for kjlu:=1 to wwfgthe -1 do
            oggtlei:=true;
        begin
         if lksswer [kjlu]> lksswer [kjlu +1] then
   begin
// look, when I’m sad, she comes to me
// with a thousend smiles
// she gives to me free
klfft:= lksswer [kjlu];
lksswer [kjlu]:= lksswer [kjlu +1];
lksswer [kjlu +1]:= lksswer [kjlu];
                            oggtlei:=false;
            end;
                        end;
                            until oggtlei;
      end.
// it’s alright, she says, it’s alright..
```

CODE, Alternative 2: Despite the unlikeliness of its layout, it results in the *same* machine code as alternative 1

Short explanation: in the majority of cases, the processor cannot read code directly. Hence, the code must be, in its turn, translated into something that the processor can read, namely machine code. This consists of the renowned ones and zeroes, but it is best not to go into much more detail. For all we need to know, the code is the last part of the programmer's actions and the first of the computer's, which makes it an interface between the worlds of human thinking and computer logic.



FIGURE, The intermediary role of code

READING CODE

This interface-nature makes of code, and coding, an object of study rich in nuances, and it has been dealt with from different perspectives. For the purpose at hand, a description of some aesthetic aspects of programming, we are particularly interested in the human perspective: code is something that must be read by programmers. Such a perspective is by no means original, but both practitioners and scientists have mostly been interest in developing methods to write code that is easier to understand. I would like to insist that this is not our concern here, instead, we want to develop an understanding of the meaning of coding styles for the programmers.

The practitioners' starting point, on the contrary, is something like "code out there, more specifically the code I have to face, is illegible" and someone should do something about it. As a result, some of them have produced their own personal sets of recommendations, also called 'coding guidelines', which generally only have local influence (within the classroom or the project at hand, for instance). In some occasions, languages come with 'official coding guidelines', which are normally modified by every programmer, or at least at every project. D. E. Knuth's coding methodology, Literate Programming (Knuth 1992), deserves special mention, and will be brought up as we move along.

The scientists' starting point is often the same, even if they perhaps do not have to face as much unreadable code. They apply however strict methodologies to their laboratory studies, trying to reach answers to questions like: how many of the programmers were able to read the code and modify it correctly? How long time did it take? What are the differences in their efficiency when one changes the coding style? One example, of many, is Paul W. Oman's and Curtis R. Cook's paper *Typographic Style is More than Cosmetic* (Oman and Cook 1990). The title of the paper is quite telling and the introduction starts like this:

> There is disagreement about the role and importance of typographic style (source code formatting and commenting) in program comprehension. Results from experiments and opinions in programming style books are mixed. This article presents principles of typographic style consistent and compatible with the results of program comprehension studies. Four experiments demonstrate that the typographic style principles embodied in the book format significantly aid program comprehension and reduce maintenance effort.

Experiments were conducted in which programmers were presented with code written according to a traditional listing and according to Oman & Cooks suggested 'book listing'. The tables show clearly that code written in the form of book listing gave better results.

TABLE 1
EXPERIMENT 1: CODE WRITING ABILITY

	Exactly correct	*Functionally correct*	*Wrong*	*Gave up or not finished*
Traditional listing (n=28)	14% (n=4)	11% (n=3)	36% (n=11)	39% (n=11)
Book listing (n=25)	36% (n=9)	16% (n=4)	32% (n=8)	16% (n=4)

TABLE 2
EXPERIMENT 2: ABILITY TO IDENDIFY PROCEDURE CALLS

Dependent measure	*Traditional listing*	*Book listing*
Number writing correct procedure	7	13
Total correct identifications	12	31
Average identifications per person	1,71	2,38
Percentage accuracy for the group	34,2%	47,6%

One could wonder whether what the results show is that code written with particular care, in order to make it as easy to understand as possible, is actually easier to understand. Which may not be a perfect tautology but is rather close. In any case, I am not interested in what style does better in laboratory settings. I am interesting in understanding what coding styles mean for programmers, how they relate to them. Oman & Curtis seem to assume that the style in which code has been written is something devoid of personal attachment, only a matter of more or less efficiency. I, on the other hand, think this is a crucial mistake: code, to programmers, is not only what the compiler translates into machine code, neither is it only what they need to read in order to understand a program: it is also something very personal, a way of expression. Programmers are not, despite some of them suggesting something along those lines, poets, but the style in which they choose to write their code says a lot about them, even if one has to be a programmer – or read this thesis – to see it.

I would like to suggest that while coding, a programmer is not exclusively concerned with making it easier for other programmers, and herself to understand what the program does. Such a view regards programming as a perfectly instrumental activity, whose only meaning is to produce something that works; and evaluates software only from the functional and economic (public) perspective. This view assumes that programmers are intelligent machines capable of solving computing problems but incapable of experiencing any aesthetic feelings towards their own creations. This assumption seems to be the general view, not only of programming but also of engineering in general. And this assumption prevails despite the slowly growing body of literature that illustrates the opposite (see, for instance, (Florman 1994; Petroski 1992; Rice 1996)).

But this focus on the public aspects of programming may be interpreted differently. Perhaps the mainstream Computer Science approach is not based on an assumption about what programmers are or do but on a judgement about how they *should* act: they should limit themselves to solving the computational problem at hand and avoid any personal preferences. Now, this is a normative, even moralistic, approach to the subject, not a scientific one. This attitude's motto could be caricaturised as: "Regardless of what programmers actually do, this is how they should do it." The problem is that what they actually do may be incompatible with how they are supposed to do it.

A good example of this is the concept of 'readability'. It seems that the majority of programmers agree that readability is important. The implication, hence, is that Oman & Cook's is the proper way to approach the subject: to search the optimal coding style. But if one studies more closely what programmers actually say, one finds that consensus on readability involves only a very general sense of the concept. When it comes to the concrete cases of coding, it turns out that it is like any other sort of writing: there are many different styles and as many different opinions about what makes code readable. The more one reads programmers speak about readable programs, the more one gets the sense that readability is a concept used to express appreciation rather than a universal measure of the program's understandability. Hence, there might be no point in going to the laboratory to search for the optimal coding style, since readability is neither solely a matter of understanding the code quickly nor a concept that can be separated from real programming environments.

Having said this, I would like to clarify that this chapter is not primarily about readability but about the alternatives available to programmers. As mentioned earlier, coding is not the only programming activity that offers alternatives to programmers, in fact, these alternatives are sometimes derided. There seems to exist a general feeling that coding is not as worthy as designing (the structure of the program), and it is possible that readers with programming skills would dismiss some of the following examples as meaningless details. But these are not included here in order to explain what programming is about but to illustrate the fact that programmers must choose between different alternatives. This fact is what gives rise to the private aspects of programming, since the choice cannot be based on calculations but require personal engagement.

When asked about it, programmers frequently explain their coding choices on the grounds of increasing readability ("I did this so and so because it makes the code more readable”), which makes it sound as if there is no personal engagement needed. However, as I have already said, there are different opinions as to what makes code readable and, hence, what we have is a personal engagement at the level of what one chooses to believe (see the discussion on instrumental beliefs in the previous chapter).

Moreover, some of the coding choices cannot be explained in any instrumental (or pseudo-instrumental) terms, making the possibility of private aspects of programming even more evident. Which brings up a small but interesting last reflection before we get into the technicalities of coding: the mere fact that programmers must make (non-calculable) decisions does not imply that there are private aspects of programming. The private aspects of programming, as has already been mentioned, require a personal engagement with one’s code. This engagement is only possible if there are non-calculable choices to be made, but it is perfectly possible that programmers simply made a choice without giving it much thought. I imagine there are programmers who are not concerned by the choices they make, as long as the code works. This is particularly true of some of the alternatives presented here, which are simple enough not to require long technical explanations but that are too simple to engage all programmers. For instance, I am quite sure that many programmers do not care much about how many empty lines they leave between functions. Enough introduction, let us see what programmers can do.

ALTERNATIVES WHEN CODING

The fact that the code must be compiled to be run, and that the compilation is a process in which all the typographical details disappear, makes coding a phenomenon with many stylistic possibilities. One can do all sorts of things with the code, as we will shortly see, and still have the same program after compilation. This means that one can look at coding from the human side instead of the processor's. In fact, the general opinion is that code should be readable. As Knuth, a famous programmer to be introduced in a few pages, put it (Knuth 1983):

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

There is one very good reason to why code should be readable, namely that it is often the only document that a programmer can resort to when faced with the maintenance of old programs.

Even if the process of programming includes many different activities (writing on pieces of paper, discussing with the end-user, arguing with the manager, being inspired by other programmers' solutions, reading books, weighing similar alternative solutions, making decisions, sometimes, perhaps often, based on incomplete knowledge, defending one's own ideas, etc.); the code is often the only trace left of all that process. Hence, the code is the only information a programmer has access to when trying to understand someone else's program – or one's own program a few months later. There should be other documents to accompany every program, but in many cases, there are not, or they are not very helpful. This is certainly not a desirable situation, but it nevertheless seems quite common, if one is to trust the literature on the subject. Why this is so is open to debate, even if the endemic lack of time in programming projects, and the fact that accompanying documentation do not make the application run, may have something to do with it. Besides, writing code and writing documents are two very different activities and it is not certain that programmers are particularly excited about the latter.

But I am not interested in finding out why those documents seem to be missing. What concerns me is the fact that, since they are missing, or incomplete, programmers can (essentially) only turn to the code when they need to figure out a program: what were the alternatives faced by the original programmer, why were technical decisions made, why the program contains the functions, variables, subroutines, objects, etc. it contains. To understand all this, they can only rely in the formulation of all the ideas in the code. So, naturally, it is important that code is readable. This, however, has not prevented programmers from having all kinds of different opinions about what makes code readable, and the mixture of its significance and its different flavours makes of readability a useful concept from which to study coding alternatives.

So let us see what programmers can do to make their code more readable. In order to make this presentation accessible to non-programmers, I am limiting myself to coding alternatives that make the code look different but that do not change the ones and zeroes that will reach the processor (i.e. the compiled code). Please note that this is a very restrictive limit; it is like explaining football by showing the alternative ways to take a corner (short, long, low, high… but what about the rest of the game?).

COMMENTS

All high level languages (basically, languages whose code requires a compiler to be translated into machine code) offer the possibility of writing comments. Those are lines, sometimes words, that the compiler filters away but that may be invaluable in order to understand the code. Let us look at the first example:

```
var
  numberOfElements, i : Integer;
  list array [1..4] of String;
  listIsOrdered : Boolean;
begin
  // we initialise the variables
  numberOfElements := 4;
  list[1]:=”G”; list[2]:=”Dn”;
  list[3]:=”S”; list[4]:=”Dv”;
  // here starts the ordering
  repeat
    for i:=1 to numberOfElements-1 do
      begin
        listIsOrdered:=true;
        if list[i]>list[i+1] then
          begin
            // theyre not in order we change them
            aux:=list[i];
            list[i]:=list[i+1];
            list[i+1]:=list[i];
            listIsOrdered:=false;
              // we have changed something
              // in the list, we are not sure
              // the list is ordered
          end;
        end;
      until listIsOrdered;
end.
```

CODE, Example with comments

Any text that appears after the command “//” is a comment (this is Delphi PASCAL). As you can see, sometimes a whole line is a comment, and sometimes the comment starts after another command. At any rate, on any given line, anything written after // is a comment and the compiler will discard it. In other words, it will never reach the processor, it is only there for programmers to read. The following code, the previous without comments, produces exactly that same machine code. It looks like this:

```
var
  numberOfElements, i : Integer;
  list array [1..4] of String;
  listIsOrdered : Boolean;
begin
  numberOfElements := 4;
  list[1]:=”G”; list[2]:=”Dn”;
  list[3]:=”S”; list[4]:=”Dv”;
    repeat
      for i:=1 to numberOfElements-1 do
        begin
          listIsOrdered:=true;
          if list[i]>list[i+1] then
            begin
              aux:=list[i];
              list[i]:=list[i+1];
              list[i+1]:=list[i];
              listIsOrdered:=false;
            end;
        end;
      until listIsOrdered;
end.
```

CODE, Example without comments

This piece of code does not include any comments, hence, the programmer that has to understand this code must rely only on what the processor is told to do. Ok, in this particular simplistic example, it is really very easy to see what the program is about, but most useful programs are thousands of lines long. If the code does not include any comments, the new programmer will have to approach it without any extra aid from the original one. Clearly, comments sound like a good idea – even if that does not mean that programmers actually do comment.

What is beauty? by **EJB** (#483024)
The Linux kernel source certainly shows some smart, lean code. But it is so lacking of commentary in the source code that it's unbelievable. It seems the whole idea is that if you can't figure out what the kernel does by looking at the C statements, you're not worth working with the source anyway.
[...] I personally prefer source code that is doesn't hurt the eyes when you look at it a few weeks after the previous time: many comments, empty lines and whitespace inserted to identify logically seperate parts of the code, etc, etc.
Erwin

The first, and most obvious, possibility for programmers to express themselves, in the sense of stamping their own style to the program, is through the text of the comments: concise and clear descriptions can change the whole feeling of a program. Apart from that, it would seem, at first sight, that there is little more to do here: comments fill a function, and it is perhaps incorrect to speak of beauty: the more comments, the better.

However, as usual, things are not so simple in reality as they seem in theory. How many comments are a reasonable amount and when does it become too much? And could it be that just inserting in a few comments lulls programmers into believing that their code is understandable? And are comments always updated when the code is modified, or are they left behind becoming a source of confusion instead of explanation? Some programmers think more about the negative than the positive effects of thorough commenting:

Re: The best code has lots of comments by **chris.bitmead** (#483063)
I have to strongly disagree too. For the argument against commenting see Extreme Programming. If you can't understand your code without comments, REWRITE THE CODE AND ADD UNIT TESTS. I can't emphasise this enough. If the code is obscure the chances the code is bad or the variable names not descriptive or the design is poor. I'm not saying don't use comments. I'm just saying only comment what is necessary, but try fixing the code first instead of applying the band-aid of comments.

Re: The best code has lots of comments by **leo.p** (#483129)
remember code is CODE, it's meant for a stupid machine, not an intelligent human), maintainability, etc.
This is bullshit. The best code is only very sparsely commented, relying instead on the clarity of its design, its data structures and it's use of the language, itself. Your principal form of documentation should be the CODE itself.
I loathe reading heavily commented source, especially in OSS [open source] projects which are rife with barely literate morons, endlessly cutting and pasting each other's code and further obfuscating it with every turn as they propagate subtle misunderstandings down the line. You very rarely should have to tell someone what you're doing, merely showing it should be enough. If you have a clever algorithm, isolate it in a single source file, document it in words and psuedo code at the very top, then show me the money.
It is a mistake to comment for rank beginers begin they cant do shit with the code, anyway. Document for people who are in a position to actually use and modify your code. Over commenting also makes it harder to understand what the fsck it is you were trying to do when the comments fall out of synch with the code and/or contain various prose and typographical errors.
Again, comment sparsely, code well. A one or two sentence description of the function, its arguements, and an enumeration of the global variables it modifies. Most everything else belongs in a man page, in its _specification_ (something that is almost always an afterthought in Linux.)
If you cant do this with your functions, your code is badly designed and no amount of commenting will fix that [...]

These last entries formed part of a longer thread in which the not only the pros and cons of commenting but also the difficulties of doing it well were discussed:

Re: The best code has lots of comments by **mckyj57** (#483178)
Good commenting style is as difficult to develop as good coding practices (the two really go hand in hand). Mental discipline (did you ever say to yourself "I'll go back and comment it later"? Did you?), clear exposition of an algorithm (no, the code is not a clear exposition -- remember code is CODE, it's meant for a stupid machine, not an intelligent human), maintainability, etc. Comments should be written in complete sentences wherever possible.

Programmers seem all to share the generally accepted but vague view that readability is a positive thing, but no clear description of what it is that makes code readable. That readability is a good thing, in general, is what we called an instrumental belief. The concept of readability connects a subjective experience of code with the usefulness of the application. The subjectivity of 'readability' is well illustrated by the different opinions held about the effects of code-commenting.

There follows now a more in depth description with more examples about the possibilities open to coders, that will not only reaffirm that programmers do not agree on how to make code more readable but will also give more details about how programmers relate to this subject in particular, and to programming in general. The following three coding issues are known to raise disputes among programmers: indentation, naming and empty lines.

INDENTATION

There is no question about whether there should be indentation or not: there should be, everyone agrees, since it does make the code more readable. But how much one should indent and how the indentation should be typed (with spaces or with tabs?) is not as clear. I shall limit myself to the discussion of the second issue, since it is somewhat different from the question of comments. The following exchange forms part of a large exchange that is presented in its entirety in Appendix 3, we focus here on the disagreement between participants *maw* and *Tassach*:

Re: The most beautiful piece of code... by **maw** (#483068)
You shouldn't use tabs in code. (The exception that makes the rule is that Makefiles require tabs.)
It's better, in a cross-platform portability way, to use individual spaces. If somebody is using an editor which can't automatically change the number of spaces, too bad for him.
Obviously, indentation is important.

Re: The most beautiful piece of code... by **Tassach** (#483209)
You shouldn't use tabs in code. (The exception that makes the rule is that Makefiles require tabs.)
Ah, yet another holy war, right up there with vi vs emacs. Personally, I hate working on code indented with spaces. I'll admit that it's annoying to edit tabbed code on a broken editor; but the way to solve that problem is to fix the editor, not the code.

Re: The most beautiful piece of code... by **maw** (#483069)
Ah, yet another holy war, right up there with vi vs emacs
No, I maintain that it is not a holy war: holy wars always concern personal preference; the tabs vs spaces debate is one of technical interoperability.
...but the way to solve that problem is to fix the editor, not the code.
I disagree, and rather than repeat the arguments myself, I point you here [http://www.jwz.org/doc/tabs-vs-spaces.html].

An editor is a program that helps the programmer write code, something not unlike a word processor, and very much like the case with processors, there are different types of editors: with or without syntax checking, with different colour codes and, as the exchange makes clear, with different behaviour regarding series of empty spaces. *maw* clearly thinks that those editors that do not "automatically change the number of spaces" (so that the code is always correctly indented) are to be dismissed, but *Tassach* seems to be of the opinion that it should be the other way round: use tabs and fix the editors that do not work well with them. I personally do not know which one is better, I do not even know how to decide, objectively, which one is the best. But the point is not to find a final solution to this issue; the point is that even details as minimal as these have importance. In my private discussions with programmers, some of them will declare themselves total fans of, say the editor 'vi', and refuse to do anything on 'emacs' (both mentioned by *Tassach*): they will explain that it is a lifestyle, and that the issue of 'vi' vs. 'emacs' is only a part of it: they learnt to code on one kind of programming environment – represented by, for instance, vi or emacs – and they identify themselves with it.

It is not that programs coded on emacs have, or lack, special functional traits that can (not) be achieved with vi. All C programs can be written in anyone of them, the language C (which is what those editors are generally used for) is absolutely independent of them. So we see that coding styles include details as trivial – for an outsider, and for the processor – as the kind of typing elements with which to make the indentation.

NAMING

Another way of improving the readability of one's code is through appropriate naming. The spelling of the commands available in a language is decided once and for all at the moment of designing the language and, in principle, the programmer can only accept the existing words. However, code also contains elements (functions, packages, variables, types, macros, classes, etc.) that are created by the programmer for that particular program and whose names, contrary to what the case is with commands, must be given by the programmer. Understanding these elements (what a function does, what values a variable stores, what sort of objects a class contains) is fundamental to interpret the code rightly, and the most natural way of making them comprehensible is by choosing a good name. Naturally, opinions vary as to what is a good name, and quite a number of naming conventions have sprung up (generally several for each language). The following example comes from the Java Programming Style Guidelines (Version 3.0, January 2002, Geotechnical Software Services Copyright © 1998-2002). Its chapter on naming conventions starts like this:

3 NAMING CONVENTIONS

3.1 General Naming Conventions

2. Names representing packages should be in all lower case.
```
mypackage, com.company.application.ui
```
Package naming convention used by Sun for the Java core packages. The initial package name representing the domain name must be in lower case.

3. Names representing types must be nouns and written in mixed case starting with upper case.
```
Line, FilePrefix
```
Common practice in the Java development community and also the type naming convention used by Sun for the Java core packages.

4. Variable names must be in mixed case starting with lower case.
```
line, filePrefix
```
Common practice in the Java development community and also the naming convention for variables used by Sun for the Java core packages. Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration Line line;

FIGURE, Naming conventions according to GSS

But as I said, there are other naming conventions and it is commonplace to complain about how each programmer has his or her own receipts. There are, naturally, disagreements about which convention to use. The names used in the example are underlined:

```
var
  numberOfElements, i : Integer;
  list array [1..4] of String;
  listIsSorted: Boolean;
begin
  numberOfElements := 4;
  list[1]:=”G”; list[2]:=”Dn”;
  list[3]:=”S”; list[4]:=”Dv”;
    repeat
      for i:=1 to numberOfElements-1 do
        begin
          listIsSorted:=true;
          if list[i]>list[i+1] then
            begin
              aux:=list[i];
              list[i]:=list[i+1];
              list[i+1]:=list[i];
              listIsSorted:=false;
            end;
        end;
    until listIsSorted;
end.
```

CODE, Variable names are underlined

I have decided to start always with a low-case letter and then use high-cases for every new word in the name (numberOfElements). But I could also have written NumberOfElements, number_of_elements, number-elements or any other possibility I could think of (within the limits set by the language – PASCAL, for instance, does not allow spaces in names and has a limit on the number of letters one can use for a name). Which one makes the code more readable? It does not make any difference to the processor since variable names disappear in the compilation.

Also notice that I have used mostly only names that mean something (list, numberOfElements, etc) but I could have shortened them: l, nE, NOE, etc.. Or called them other things: series, nElements, numberE, listing, catalogue... Notice also how I use the letter ‘i’ as a name for a variable. This variable is a loop-counter, it increases every time the loop is passed through and it is an auxiliary variable. Since it only is meaningful in the local context of the loop, programmers often use just one letter to name them. However, it is not clear whether one should use one-letter variables at all. Many guidelines recommend it, but not everyone agrees. In order to understand the following exchange it is important to know that some variables are used throughout the whole program (global variables) whereas others are only used temporarily in short parts of it (local variables). An example of local variables are the counters and indexes used in loops, such as my 'i'.

#1 Peeve by **Anonymous Coward** (#2253422)
I know the man [D. Knuth] is brilliant, but in my opinion his code samples are CRAP. I know this will get moderated as flame bait, but it comes from the heart. One letter variable names should be OUTLAWED from languages. I don't know how many times I've had to wipe up the shit that someone left because they didn't take the time to think up better variable names. Second to this is variable names are that are sequenced. For example: a1, a2, a3.No doubt this will generate a lot of heat, but I know there has to be others who feel the same way.

Re:#1 Peeve by **Anonymous Coward** (#2253785)
One letter variable names should be OUTLAWED from languages
Except for counters or indexes in loops. Anything more than i, j, or k is overkill and actually distracts from readability

Re:#1 Peeve by **Anonymous Coward** (#2254080)
Except for counters or indexes in loops. Anything more than i, j, or k is overkill and actually distracts from readability
ESPECIALLY in counters indexes and loops. Everytime I hear someone mention these exceptions, I PUKE!!! Those harmless counters and loops always find their way into the program somehow, outside the scope of their original domain. If I had my way, they'd be outlawed without exception. And since I'm the programming manager, I get my way. No one in my shop gets to use one letter variables.

As explained in Slashdot's introduction (*method and empirical material*), whenever a user wants to insert a post without filling in his or her name (or alias), the post appears as written by *Anonymous Coward*. So the previous exchange is, most likely, not written by a programmer with multiple personalities but by two – or three – programmers who did not sign their entries. Once again we can see how technical issues get discussed in aesthetic terms ("I PUKE!!!"), perhaps this particular outburst is a response to the fact that the general opinion seems to be that counters and indexes in loops should be given one-letter names. Not only does D. Knuth use them in his programs, most guidelines recommend them (this is one of the recommendations of the above introduced Java Programming Style Guidelines):

22. Iterator variables should be called i, j, k etc.

```
while (Iterator i = pointList.iterator(); i.hasNext( ); ) {
  :
}

for (int i = 0; i < nTables; i++) {
  :
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators.

Variables named *j*, *k* etc. should be used for nested loops only.

FIGURE, Naming conventions according to GSS

Here, as in the case with comments, the programmers' inventiveness and skill in finding appropriate names gives the program a personal flavour. Even if one concrete convention was adopted by all programmers – unlikely – there would still be room for personal expression in the choice of names. Needless to say, as the title of a book, or the choice of chapter numbering, influence the reader's aesthetic experience, the names of the variables also have an influence in the programmers' opinion of the code.

Another option available to programmers when writing code is the introduction of empty lines and spaces, either to make it more readable or just more elegant. The example, with two different empty lines - spaces policy may look as follows:

```
var
  numberOfElements, i : Integer;
  list array [1..4] of String;
  listIsOrdered : Boolean;
begin
  numberOfElements := 4;
  list[1]:=”G”; list[2]:=”Dn”;
  list[3]:=”S”; list[4]:=”Dv”;
    repeat
      for i:=1 to numberOfElements-1 do
        begin
          listIsOrdered:=true;
          if list[i]>list[i+1] then
            begin
              aux:=list[i];
              list[i]:=list[i+1];
              list[i+1]:=list[i];
              listIsOrdered:=false;
            end;
          end;
    until listIsOrdered;
end.
```

CODE, Alternative 1: without empty spaces

```
var
  numberOfElements, i : Integer;
  list array [1..4] of String;
  listIsOrdered : Boolean;

begin
  numberOfElements := 4;
  list[1]:=”G”; list[2]:=”Dn”;
  list[3]:=”S”; list[4]:=”Dv”;

    repeat
      for i:=1 to numberOfElements-1 do
        begin
          listIsOrdered:=true;
          if list[i]>list[i+1] then
            begin
              aux:=list[i];
              list[i]:=list[i+1];
              list[i+1]:=list[i];
              listIsOrdered:=false;
            end;
          end;
    until listIsOrdered;

end.
```

CODE, Alternative 2, with empty spaces

Both result, once again, in exactly the same machine code after compilation. Which one is more readable? Which one is more elegant? Which one would you choose?

CODING STYLES

Although I have not quite resolved what it is that enhances a program’s readability, this is only of secondary interest to the discussion. What is important to us is that code, apart from transmitting, more or less satisfactorily, the ideas behind the design (readability), also serves as a sign of the author’s personal preferences. Choosing an approach to readability is, in itself, a part of one’s personal attitude towards programming, of one's style. In fact, it could be argued that you do not choose an approach to readability but that it is the coding style that appeals to you that you will find readable. At any rate, it seems clear that readability is a matter of subjective opinions.

Readability and coding styles are hence at the heart at the personal relationship between programmers and software (private aspects of programming) so it should be interesting to classify them, just to put some order and to clarify some of their aspects. There are, of course, no official classifications available, but there do not seem to exist even some private ones, or at least I have not found any. Hence, I have decided to construct one such classification that uses the concepts presented above and that only has to classes:

LITERATE	MINIMALIST
comments –	no comments
empty spaces –	no empty spaces
long naming –	hort naming

Now some programmers like to have no comments and short naming and are inconsistent when it comes to empty lines; others name inconsistently but put a lot of care in the comments; there are all sorts. And I have not discussed other alternatives given by particularities of the languages – for instance the possibilities offered by the #define in C. In fact, almost each programmer has her/his own coding style, making classifications such as the one above somewhat unreal. But still, they may serve as examples of what those styles look like and how programmers relate to them.

I have called the two suggested styles 'literate coding' and 'minimalist coding'. These two names are not used by programmers, nor do they identify two classes similar to these ones; they are the result of my attempts at establishing some order to the discussion. It may, in fact, be so that some programmers recognise some of the characteristics of their code on the literate side and some others on the minimalist side. But, once again, the purpose is not to construct a final classification but to find an inroad to the private aspects of programming.

LITERATE CODING

'Literate coding' is a name directly inspired in 'Literate Programming,' a concept introduced by Donald E. Knuth in (Knuth 1983). In the introduction to that paper he described the main goal of that programming methodology:

My purpose in the present paper is to propose another motto that may be appropriate for the next decade, as we attempt to make further progress in the state of the art. I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature.
Hence, my title: "Literate Programming."
Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.
The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.

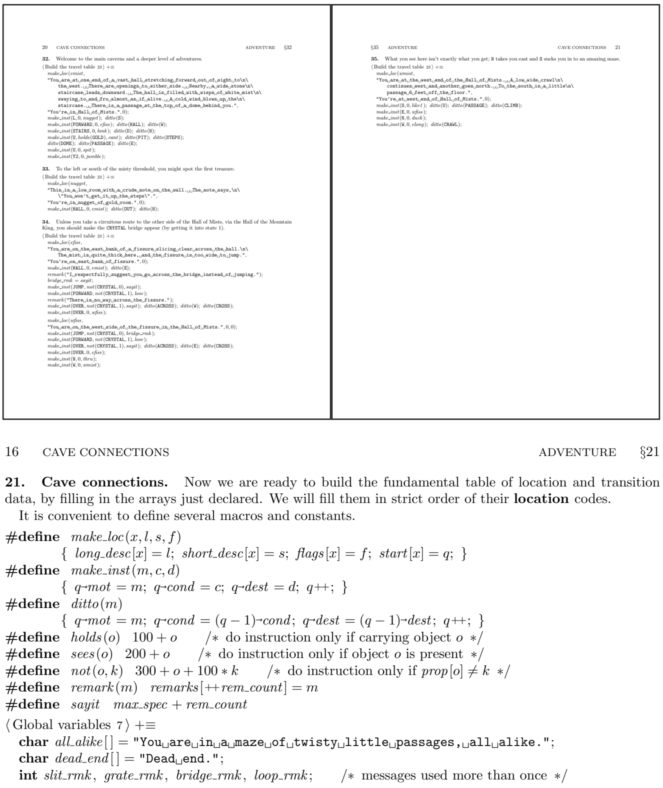
It is perhaps no coincidence that Knuth's best known publication is a series of books called *The Art of Programming* (Knuth 1997) (see more of these ideas in (Knuth 1974a)), a very technical, and much admired, work that starts with the following lines:

The process of preparing programs for a digital computer is especially attractive because it not only can be economically and scientifically awarding, it can also be an aesthetic experience much like composing poetry or music.

The literate methodology of programming, and the coding style that accompany it, goes much further than the usual guidelines that can be found a little bit everywhere. The main concern of the programmer is with exposition and *excellence* of style. Literate programming is as much a question of readability as of aesthetics, of personal expression.

Knuth is not just another coder, he received the Turing Award in 1974 (nowadays Professor Emeritus of The Art of Computer Programming at Stanford University), and is quite famous among programmers. One detail from his work may serve as an amusing illustration of the many nuances of the programmers relation to software. One of Knuth's well-known ideas is to offer money for every bug found in the programs he has published. The reward for every bug increases both with time and with the number of bugs already found and nowadays there are checks for $300 in circulation with his signature on. I do not think that they would put much pressure on his bank account but he may not even have to worry: those checks are not always cashed in, they are instead framed and kept as trophies. The whole thing has come to be called Knuth's entomology…

But what does a literate program look like? The following is an excerpt from a program that Knuth wrote in order to show the possibilities of his style. It is called *Adventure* (Knuth 1998) and it is a remake of a famous early computer game:



20 CAVE CONNECTIONS ADVENTURE §32

32. Welcome to the main caverns and a deeper level of adventures.

```
⟨Build the travel table 23⟩ +≡
  make_loc(emist,
  "You␣are␣at␣one␣end␣of␣a␣vast␣hall␣stretching␣forward␣out␣of␣sight␣to\n\
      the␣west.␣␣There␣are␣openings␣to␣either␣side.␣␣Nearby,␣a␣wide␣stone\n\
      staircase␣leads␣downward.␣␣The␣hall␣is␣filled␣with␣wisps␣of␣white␣mist\n\
      swaying␣to␣and␣fro␣almost␣as␣if␣alive.␣␣A␣cold␣wind␣blows␣up␣the\n\
      staircase.␣␣There␣is␣a␣passage␣at␣the␣top␣of␣a␣dome␣behind␣you.",
  "You're␣in␣Hall␣of␣Mists.", 0);
  make_inst(L, 0, nugget); ditto(S);
  make_inst(FORWARD, 0, efiss); ditto(HALL); ditto(W);
  make_inst(STAIRS, 0, hmk); ditto(D); ditto(N);
  make_inst(U, holds(GOLD), cant); ditto(PIT); ditto(STEPS);
  ditto(DOME); ditto(PASSAGE); ditto(E);
  make_inst(U, 0, spit);
  make_inst(Y2, 0, jumble);
```

33. To the left or south of the misty threshold, you might spot the first treasure.

```
⟨Build the travel table 23⟩ +≡
  make_loc(nugget,
  "This␣is␣a␣low␣room␣with␣a␣crude␣note␣on␣the␣wall.␣␣The␣note␣says,\n\
      \"You␣won't␣get␣it␣up␣the␣steps\".",
  "You're␣in␣nugget␣of␣gold␣room.", 0);
  make_inst(HALL, 0, emist); ditto(OUT); ditto(N);
```

34. Unless you take a circuitous route to the other side of the Hall of Mists, via the Hall of the Mountain King, you should make the CRYSTAL bridge appear (by getting it into state 1).

```
⟨Build the travel table 23⟩ +≡
  make_loc(efiss,
  "You␣are␣on␣the␣east␣bank␣of␣a␣fissure␣slicing␣clear␣across␣the␣hall.\n\
      The␣mist␣is␣quite␣thick␣here,␣and␣the␣fissure␣is␣too␣wide␣to␣jump.",
  "You're␣on␣east␣bank␣of␣fissure.", 0);
  make_inst(HALL, 0, emist); ditto(E);
  remark("I␣respectfully␣suggest␣you␣go␣across␣the␣bridge␣instead␣of␣jumping.");
  bridge_rmk = sayit;
  make_inst(JUMP, not(CRYSTAL, 0), sayit);
  make_inst(FORWARD, not(CRYSTAL, 1), lose);
  remark("There␣is␣no␣way␣across␣the␣fissure.");
  make_inst(OVER, not(CRYSTAL, 1), sayit); ditto(ACROSS); ditto(W); ditto(CROSS);
  make_inst(OVER, 0, wfiss);
  make_loc(wfiss,
  "You␣are␣on␣the␣west␣side␣of␣the␣fissure␣in␣the␣Hall␣of␣Mists.", 0, 0);
  make_inst(JUMP, not(CRYSTAL, 0), bridge_rmk);
  make_inst(FORWARD, not(CRYSTAL, 1), lose);
  make_inst(OVER, not(CRYSTAL, 1), sayit); ditto(ACROSS); ditto(E); ditto(CROSS);
  make_inst(OVER, 0, efiss);
  make_inst(N, 0, thru);
  make_inst(W, 0, wmist);
```

§35 ADVENTURE CAVE CONNECTIONS 21

35. What you see here isn't exactly what you get; N takes you east and S sucks you in to an amazing maze.

```
⟨Build the travel table 23⟩ +≡
  make_loc(wmist,
  "You␣are␣at␣the␣west␣end␣of␣the␣Hall␣of␣Mists.␣␣A␣low␣wide␣crawl\n\
      continues␣west␣and␣another␣goes␣north.␣␣To␣the␣south␣is␣a␣little\n\
      passage␣6␣feet␣off␣the␣floor.",
  "You're␣at␣west␣end␣of␣Hall␣of␣Mists.", 0);
  make_inst(S, 0, like1); ditto(U); ditto(PASSAGE); ditto(CLIMB);
  make_inst(E, 0, wfiss);
  make_inst(N, 0, duck);
  make_inst(W, 0, elong); ditto(CRAWL);
```

16 CAVE CONNECTIONS ADVENTURE §21

21. Cave connections. Now we are ready to build the fundamental table of location and transition data, by filling in the arrays just declared. We will fill them in strict order of their **location** codes.

It is convenient to define several macros and constants.

```
#define make_loc(x, l, s, f)
          { long_desc[x] = l; short_desc[x] = s; flags[x] = f; start[x] = q; }
#define make_inst(m, c, d)
          { q→mot = m; q→cond = c; q→dest = d; q++; }
#define ditto(m)
          { q→mot = m; q→cond = (q − 1)→cond; q→dest = (q − 1)→dest; q++; }
#define holds(o)  100 + o      /* do instruction only if carrying object o */
#define sees(o)  200 + o      /* do instruction only if object o is present */
#define not(o, k)  300 + o + 100 * k      /* do instruction only if prop[o] ≠ k */
#define remark(m)  remarks[++rem_count] = m
#define sayit  max_spec + rem_count
⟨Global variables 7⟩ +≡
  char all_alike[] = "You␣are␣in␣a␣maze␣of␣twisty␣little␣passages,␣all␣alike.";
  char dead_end[] = "Dead␣end.";
  int slit_rmk, grate_rmk, bridge_rmk, loop_rmk;      /* messages used more than once */
```

FIGURE, Knuth's Adventure: .iterate code

As you can see, the code of the program is structured like a book, with conventional page numbers. More specifically, it looks like a law book with all the text divided into sections. This is section §21 (page 16), there are 201 for this program in particular, all of them with their 'text-introduction' and their piece of code (I wonder if Knuth would rather have me saying "everything is code"). Literate Programming is probably as far as anyone can go in the direction of explicitness in coding style. And it has its share of admirers in the Slashdot discussions about beautiful software:

some suggested resources by **Satai** (#2252829)
Personally, I found Donald Knuth's [stanford.edu] Literate Programming [amazon.com] as well as the Practice of Programming [amazon.com] to be wonderful resources for writing better, more beautiful code.

Personal recommendation by **mattbee** (#483047)
People seem to be mentioning the obvious targets: Knuth, BSD etc [...]

The best code has lots of comments by **Dr. Tom** (#483060)
One of the reasons people like Knuth's approach so much is that he puts the comments first, conceptually. The code is essentially embedded in a great long comment that describes everything that's happening. The code is just there to distill the essence of the algorithm into a form a stupid machine can understand. If the machines were a bit smarter, they would be able to run the program by reading the comments and executing them!

TEX by **protek** (#483152)
If TEX by Donald Knuth doesn't bring tears to eyes nothing will. ;-) But seriously, there probably isn't a better example of programming at it's finest, particularly if you are interested in Literate Programming

To the literate coding style belongs the code written with extensive comments, with generous use of empty lines and empty spaces, with long descriptive variable names, and so on. It is a sort of style in which care is taken to use all the available typographical options to make the design as explicit as possible.

MINIMALISM

It may seem at first sight that 'literate coding' is the coding style that really tries to make programs readable while the opposite is writing unreadable code. But this would be incorrect; we saw earlier in the section that there are programmers that do not particularly think that extensive commenting yields, for instance, more readable code. On the contrary, they seem to say that people that use comments profusely are programming in a rush and that code should be so carefully written that it is “self-documenting”: “I think code should be good enough to stand on it's own without comments.” Furthermore, they argue, comments can be outdated by modifications to the code and, instead of help, they might confuse the programmer:

Re:The best code has lots of comments. by **alvi** (#483151)
[...] First of all, if someone is supposed to maintain a piece of code, he or she has to read the code (not the comments!) and understand it. Period. In most of the cases, its even better to just forget about the comments at all. Comments won't be translated into machine code... but the source code will, and that's what's going to run in the end. It's unlikely that a programmer will adjust all the comments (if there are changes in the code) to be completely consistent all the time. You simply can't trust comments.

This minimalist style is based on the idea of coding as an austere translation of the design. They also argue that comments take up space on the screen, preventing an overview of the actual code (what the processor will read):

The best code has comments only when needed by **a!b!c!** (#483208)
Its interesting that you mention how to format comments but not what you put in them. Excessive and useless commenting is almost as bad as having no comments at all. Its frustrating to have the code so chopped up with crap, that I can't even fit 15 lines of code on the screen. Good variable names can greatly reduce the need for comments.
[...] I greatly prefer simple code that can be easily read with minimal comments.

I cannot find a name quite as illustrious as Knuth to support this side of the argument, but Robert Pike is sufficiently eminent. He is (2003) a member of Technical Staff at AT&T Bell Laboratories in Murray Hill, New Jersey (as Knuth was) and he co-authored with Brian Kernighan *The Unix Programming Environment* (Kernighan and Pike 1984), among other things. He has published some *Notes on Programming in* C (Pike 1989) on the internet in which he expresses his opinions about comments:

Comments
A delicate matter, requiring taste and judgement. I tend to err on the side of eliminating comments, for several reasons. First, if the code is clear, and uses good type names and variable names, it should explain itself. Second, comments aren't checked by the compiler, so there is no guarantee they're right, especially after the code is modified. A misleading comment can be very confusing. Third, the issue of typography: comments clutter code. But I do comment sometimes. Almost exclusively, I use them as an introduction to what follows. Examples: explaining the use of global variables and types (the one thing I always comment in large programs); as an introduction to an unusual or critical procedure; or to mark off sections of a large computation. There is a famously bad comment style:

```
        i=i+1;              /* Add one to i */
```

and there are worse ways to do it:

```
         /**********************************
         *                                 *
         *          Add one to i           *
         *                                 *
         **********************************/

                        i=i+1;
```

Don't laugh now, wait until you see it in real life. Avoid cute typography in comments, avoid big blocks of comments except perhaps before vital sections like the declaration of the central data structure (comments on data are usually much more helpful than on algorithms); basically, avoid comments. If your code needs a comment to be understood, it would be better to rewrite it so it's easier to understand.

There is no term that is generally used to describe this style of coding but 'Minimalist' is, I believe, appropriate. Minimalism, as I see it, is not only about sparse commenting, it is also about strictness in the adherence to the grammatical rules of a programming language. It is against all kinds of 'clutter'.

Pike says that "comments clutter code" and he means that since code is not executed by the processor, they are in the way when reading the code. In order to understand what the processor does one should focus on the parts of the code that the processor actually 'reads'. But 'clutter' can also be used to name unnecessary commands or overzealous use of language:

Re: The most beautiful piece of code... by **RoninM** (#483167)
How about:

```
#include <stdio.h>
int main(void)
{
   (void)printf("Hello, world!\n");
}
```

Re: The most beautiful piece of code... by **joto** (#483199)
No, it's not beautyful.
[...] 2. It is stupid to cast the return value from printf(). It introduces more visual clutter, and serves no purpose.
3. I think you could afford a line of whitespace between the preprocessor directive and the main function. [...]

The second point in *joto's* reply refers to the *(void)* that *RoninM* has included before the *printf("Hello, world!\n")*. I think a short technical explanation is necessary here: functions are elements of most programming languages which have precisely defined rules of use. Those grammatical rules, as they are called, have to be strictly followed when writing code if one wants the compiler to be able to translate it into machine code. However, sometimes the rules might allow for slightly different uses, generally to simplify the writing of programs. We have an example in the previous exchange: functions in C always return a value when executed (which technically means that the result of the operations carried out by it will be stored in a particular place of the memory) however some compilers allow function calls that do not use the value returned. This was probably included in some versions of C in order to improve the ease of use of functions that carry out interesting operations but whose return value is generally uninteresting. *printf()* is one of those functions: when called, it writes a given string of characters on the screen, if everything goes right it returns the value TRUE otherwise it returns FALSE. I do not exactly know in which conditions *printf()* can fail (probably depends on the compiler and the operative system) but they are certainly rather rare. Which means that many programmers do not even bother to check the return value.

RoninM's program does not exactly check the return value, the code just makes sure that the value is cast away under controlled forms, that no ends are left loose. Some programmers, like *joto*, find the whole thing redundant and confusing, since there is no strict need of casting in this particular case:

Re: The most beautiful piece of code... by **joto** (#483203)
[...] Anyway, I think any C-programmer on the planet knows that printf() is called mainly for a side-effect. You do not need to tell them that with a void-cast, as little as you need to tell them that with a comment. Do you really think there is even a single programmer on the planet that think it is easier to understand your programs because you put in lots of redundant unnesseceary casts?

This exchange between *RoninM* and *joto* shows two different approaches to grammatical rules: the rigorous and the relaxed one. I have decided to include the rigorous one in the literate style because it can be considered a form of clutter, and minimalists are against anything that may appear on the computer screen and has no significant influence in the way the processor executes the code. However please note that the 'relaxed' approach to grammatical rules in programming is quite far from what 'relaxed' generally means, the rules do not offer many alternatives and compilers are perfectly intolerant, anything that is not explicitly allowed in the rules will stop the compiling process.

OBFUSCATED CODE

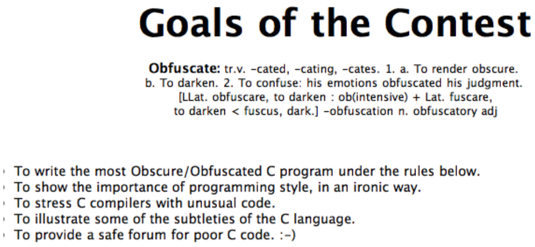
The previous sections present two different ways of relating to code. They do not exactly mirror the situation among programmers (they consider more aspects than our three) but serve as an illustration of how coding involves more than a search for readability. Personal preferences play an important role, if anything, because it is impossible to decide a priori what kind of coding styles are more readable.

I would like to wrap up this chapter on coding styles, and their connection with the private aspects of programming, with a short presentation of a curious programming phenomenon: obfuscated code. Obfuscated code is the opposite of readable code, the idea is to write programs that are impossible to understand. Some programmers joke(?) that they are forced to do so at their working places in order to make sure they will not be sacked:

Re:True, but code maintainability can be critical. by **Anonymous Coward** (#2253176)
*There's (usually) no guarantee that *you* are going to be the one maintaining the code in the future*
I thought that was the point of writing sloppy code in the first place!
Job Security: it's a wonderful thing ;-)

Re:Open Source, of course by **Telek** (#2252952)
[...] Mind you, at one place that I worked where our jobs weren't very secure I used obscurity of code to secure my job. Noone else in their right mind could understand what I wrote (not necessarily due to messyness, but no comments, and not meaningful variable names, no documentation, etc). I had written the entire application, and before I left I commented it, but it was fun at the time.

But generally, programmers write obfuscated code (at least consciously) in order to enter contests. Perhaps the most famous of this is the International Obfuscated C Code Contest (IOCCC), to which people send all kinds of inhuman, and technically extremely advanced, creations. The IOCCC is held under very relaxed forms, there are no prices other than peer-recognition, and its internet site presents a rather humorous tone. These are the official goals of the contest:



Goals of the Contest

Obfuscate: tr.v. -cated, -cating, -cates. 1. a. To render obscure. b. To darken. 2. To confuse: his emotions obfuscated his judgment. [LLat. obfuscare, to darken : ob(intensive) + Lat. fuscare, to darken < fuscus, dark.] -obfuscation n. obfuscatory adj

- To write the most Obscure/Obfuscated C program under the rules below.
- To show the importance of programming style, in an ironic way.
- To stress C compilers with unusual code.
- To illustrate some of the subtleties of the C language.
- To provide a safe forum for poor C code. :-)

FIGURE, IOCCC's goals

But the real goal of the contest is to show who can write the most amazing piece of code (making code really unintelligible requires some skills). Every year programmers send in their programs, and every year one can find quite impressive examples of code. Let us look at one of them.

A few introductory words first: every submission to the IOCCC must be accompanied by a description of the methods used to obfuscate the code, in other words, what the author has done to make the program as incomprehensible as possible (often the obfuscation, as the case is with our example, simply results from squeezing the program into the size allowed by the contest). Bernd Meyer, one of the winners (for "Best Utility") in the 2000 edition, says the following about the obfuscation in his glicbawls program (I know it might feel a bit too technical, but the effort is well worth it):

C) Obfuscation
==============
Most of the obfuscation of this program was driven by the desperate struggle to fit all of glicbawls' features into code that fits the IOCCC's size limits. All identifiers names are single-character, which was only possible by ruthless exploitation of scope. There is a function M() that handles nearly all input from stdin, handles all output to stdout, and also recursively calculates one of the mathematical expressions needed for error modelling. There is a function that calculates (A^(-1)*b)*x with A a matrix and b and x being vectors, and does so in a very efficient way --- but one would be hard pressed to find it. There is an arithmetic coder that is even less recognizable. A lot of loop variables don't get initialized, but instead the code is arranged in such a way that they just happen to have the right value when control flow reaches the loops. Lots of functions use the same two global scratch variables. Variables change their meaning all the time, and the differences between compressing and decompressing are handled in rather subtle ways all over the program. The string that holds the ascii values for displaying the progress image doubles as a description on how to find a pixel's causal neighbours. And then there are a few #define's that really are just in there to save a few characters, but as an added "benefit" make the source ever more unreadable.

In short --- running it through the preprocessor and the pretty-printer will give you something that looks slightly less like line noise and slightly more like a C program, but unless you are a true wizard, it is unlikely to gain you any insights into what is actually going on....[o]

Bernd Meyer

In case you are wondering what the code looks like, this is it (in very small font size but it does not really matter, it's quite indecipherable in any size):

```
#include                 <stdlib.h>
   #include                 <stdio.h>
#define           a         typedef
a         long                N;
#define         i =m(p(r,o,v),e,d
N         G,l,I,C,B,A,W,L,S
,       R,O,c,k,s             =
80,                    U=13
,T=169;                   a double P;
   #define               F for(
      a P*E; a char*w;               P sqrt(P); N H(){ F;
      O=scanf(" %[#]%*[^\n]",&O); ); scanf("%ld%*c",&O);
          #define D return
          D O; }



#define V             =malloc(sizeof(
#define  M             M(
#define   X           +=
P M N R){ s=                 s>=k?printf( "%ld%c"
+3*(C<1),s-k,               R&7?32:10):s; D R>0?B
/2?H():getchar               ():R?.9+.7*M R+1):0; }
P t(N x,P K){ D             x?(x>U?0:t(x+2,K)*(x-1)
/x/K)+1/sqrt(K):           t(2,K*K/U+1)*K; }  N _(N
J,N x){ F*(x?&W:&         A)=W+J+x; (O=x=W/(J=k*k))
||A<J||(( O=2*W/J)         &&2*A/J<3); A    X A-J+1){ F
J*=x+O; S--*B&&x==O       ; O=x=!J){ s X x+s; M 0); }
W X W-J;  S X 2; L X             L-J*k+M S%8==B); } D  1; } E
p(E J,E x,E O){ F R=T;            R--; )O[R]=J[R]+.8*x[R]; D O;
} P m(E W,E x,P s){ F R    =         U; --R; ){ E A=W+R*U; W[R]X s/
12; *W X s; *A-=*x++; F;  A>W;      ){ *A/=*W; F O=R; O; O--)A[O]-=
W[O]**A;  A-=U; } W X 14;     } D*W; }     int main(int z,w*y){ N K=s,g; Y;
M z); s= B=C=15-H(); s=G=   M 8); s=l=M  C); G*=g=C%3?1:3; if(I=s=M 8)){ N
b=G+9,x=b*340+U,J=b*(l+5), * n V N ) * J ),*j=n+5*b; E h V P)*x),q=h+b,e=q+
b; P d=K; F; J--; )n[J]=J<b ? 0: I / 2  ; F; x; )h[--x]=0; s=M x); k=256; C
=4-C; B=C/4;  s=_(_(T,0),0); F ; -- z  ; ){ R=atoi(*++y); c=R>0?K=R,c :-2*R;
} z=G/K+1; z X g>z; I++; c++; F ;  ++ J<l; ){ P u=0,C,y; E H =e+U,o= H+b*T,r
=o-2*T,v=r-T; F; x<G; x++){ p( H , o -T,o); o X T; H X T; q[x]= .7*(q[x-1]+h
             [x] );
             } p(H,
             H,r );
             F;  x;
            ) { E S=
           e ; w  l =
        " !{   ,;1f6D@"
      ; j X 1-g; F; *++l;
          ){ *S++ =*j; j X*l%3*g-
        g+*l%5*b-3*b; } y =M  x-G);
      y =M-J)* M-x)+ M-J-1)*y+.01; y=
    sqrt((u+q[--x]+.1)/y ); o-=T; C i)+
        #define Z(x)(t(0,(x-C)/y)-l+1e-6*(x-Q))
      .5; K=M-B); { N f=I,Q=0; F; f-Q>c; ){ P l=0
    ; N H = C+c/2+1; H%=c; H X(f+Q+c-2*H)/2/c*c; l=
  Z(Q); O=(A-W)*Z(H)/Z(f); _(O,R=B?K>= H:L/k>O+W); *(
      R?&Q:&f)=H; } Q X f; *S=*j=Q/=2; s=B?s:Q+k; f=Q+n[x/z];
    n[x/z]=x%z+J%(z*2/g)?f:!putc(x?l[4*g*f/z/z/I-8]:10,stderr);
  H-=U; F O=156; O--; ){ H--; *H=e[O%U]*e[O/U]/y+.8**H; } C-=Q+.5
; y i/.9)-Q-C; p(p(r+T,r,r),H,r); h[x]=.7*h[x]+C*C; u X h[x]; u*=.7
       ; d X C>0?-y:y; } } j X 9; } _(_(x,x),x); } D 0; }
```

CODE, Meyer's creation: glicbawls

This program compresses and decompresses images, but it might be close to impossible to realise this just by looking at the code. Meyer has gone more or less as far as you can go in trying to make programs unreadable: not initialising variables, calling them short hermetic names, using the same function to do different things, etc. The preprocessor and the pretty-printer that Meyer talks about are applications that are capable of putting *some* order in the code (so that it does not look like a Martian character, for instance) but that is, according to him, not going to help you much. This is the paragon of unreadability, and this extreme sort of programming is not without admirers:

Beautiful... sort of ! (beautiful obfuscated C) by **Cedricm** (#483054)
[where can I find beautiful code?] See http://www.ioccc.org/
Cool stuff !

This is not off-topic. Lousy moderation. by **BeanThere** (#483073)
Some of the most "elegant and masterful design" I've ever seen is code from the obfuscated C coding competition (http://www.ioccc.org/; it may often look pretty atrocious to the "untrained eye", but there are some pretty amazing examples of masterful design [...]

Want nice code? by **alehmann** (#483100)
http://www.ioccc.org [...]

Try this.. by **Arjuna Theban** (#483220)
Check out savastio.c or any other ioccc winner.
If you can read those, well.. you're either too good or have too much time in your hands. Either way, it's good.

Even if they don't explicitly put it like this I believe they would all admit that it is unreadable, but that is hardly a point when the title of the contest is "International Obfuscated C Code Contest." The point is that it shows the wide range of alternatives open to programmers when coding, and also that readability must sometimes be weighed against the pleasure of writing something with a personal touch, even if it need not go to the IOCCC extremes. The concept of vanity is not alien to programmers:

Re:Beautiful software by **quartz** (#2253179)
I never said I blamed Perl. I just said it's what I happen to be coding in at the moment, and I'm loving every minute of it. :-) I know I should strive to make my code "readable", but the irresistible urge to type:

join(" ", map { ucfirst } split(/ /, shift)); just to correct a spelling error in the database on-the-fly is simply overwhelming. I just can't help myself. :)

Re:Beautiful software by **fishbowl** (#2253306)
I know I should strive to make my code "readable", but the irresistible urge to type:join(" ", map { ucfirst } split(/ /, shift));
The presumption that your code is somehow "unreadable" bothers me. I find your transformation to be okay, although I don't like the bareword ucfirst. If I were maintaining your code, I'd probably do away with your use of the $_, or at least, explicitly use $_ instead of implying it. But there's nothing in this example that should be a problem for even a beginning perl coder, in my opinion. You've used a common perl idiom in a very efficient, clear, understandable way.
Now, if an ADA or VB programmer can't understand your program, that's an unreasonable criticism.

Explanation: Perl is a language that allows for really compact code, things that are quite difficult to understand. IT is a sort of standing joke that it is impossible to write readable code in Perl, and *quartz'* first comment is “My software is so ugly it's beautiful. I'm coding in Perl these days. :).” Now, some Perl users can be sensitive about their language and one of them claimed that Perl should not be blamed for one’s own lack of skills. *quartz'* answer to that is the first of the previous comments: he or she admits that Perl is not to blame, but that he feels the “irresistible urge” to code things as complicated as “join(" ", map { ucfirst } split(/ /, shift))”. *fishbowl* reacts to this, saying that he or she does not see why on earth that line should be considered unreadable. Then follow a few comments from Perl users that explain what the different parts of the line do and how it all works. And then comes *quartz'* answer to *fishbowl*, read attentively:

Re:Beautiful software by **quartz** (#2254206)
OK, OK, you got me. The $_ was actually there when I pasted the code from the Emacs window, but just before I submitted I decided to take it out for added effect, as I know many non-Perl coders have, um,

strong opinions about implicit variables. Vanity, I guess. But hey, it does work with strict and -w!:)
As for the unreadability part, try writing the equivalent of the above in C++ (or, god forbid, Java) and looking at the 10+ line resulting code you'll see why C++/Java coders at least might find unreadable what's otherwise perfectly fine Perl.

"Vanity, I guess"...I think it is safe to assume that coding, for *quartz*, is not only about writing readable programs.

VI
Aesthetic Ideals

Through an examination of coding and its results, this thesis has introduced the technically complex subject of beauty in programming but it is not my intention to go into a detailed explanation of the aesthetic ideals of programming since this would require too much technical overhead. However, this study would be incomplete without an overview of the most popular aesthetic attributes of software. I will present here the following ideals: cleanness, simplicity, tightness, consistency, structure and robustness. Programmers use these words (the adjectives) to describe the beauty of their preferred programs, or, perhaps more often, the qualities they pursue when writing software. As in the previous chapter, the purpose here is not to provide a comprehensive classification but to delve more deeply into the phenomenon by offering more evidence of the existence and significance of the private aspects of programming.

Re:Not this stupid 'programming is art' BS again! by **Anonymous Coward** (#2255136)
Is coding by itself even a craft? It's really nothing more than translating an algorithm from one form to another. Under copyright law, I can make a case that *coding* is nothing more than a work-for-hire. I find the arguments against coding standards really lame. "It hurts my creativity".
Buddy, if your creativity depends on the placement of braces and how many spaces you use to indent, you're a real lamer.
Designing, on the other hand, *is* art.

Programming is so much more than coding that it would be misguiding to only write about coding styles when discussing software aesthetics. This chapter will introduce the different qualities that programmers appreciate, and to which they relate in aesthetic terms. These descriptions are not only offered in order to expand the picture of software aesthetics but also to further illustrate how programmers relate to software. The study's ultimate goal is, after all, to gain an understanding of this relationship. Let me first include a short reflection about the field of aesthetics and about the sense in which 'aesthetics' (or beautiful) is used here and by programmers.

AESTHETICS

The discipline of aesthetics is perhaps the most popular branch of philosophy, and the interested reader can chose among a great number of approaches to it. There is little point here in flooding the text with references to books in this field, suffice to point here to three good introductions (Hofstadter and Kuhns 1976), (Valverde 1998) and (Langer 1957).

Aesthetics can be described in a few words as the ontological and epistemological study of beauty and art (what are beauty and art and what can be known about them), was originally devoted to the study of feeling – as opposed to reason. This realm of the human mind had been mostly ignored in Western thought, overshadowed by the notion that legitimate knowledge could only be achieved through logical, devoid of feelings, rationality. In the renaissance and the seventeenth century, however, man, indeed exclusively man, became the centre of philosophical theories, displacing other classic cosmic concepts such as order, harmony or God, as the main epistemological referent. The texts that have often served to mark the turning point in this change of views is Immanuel Kant's three *Critiques*.

Once this change of focus was made, it became impossible to ignore the existence and significance of feelings, even if their place in the grand scheme of things was not (still is not) obvious. Many philosophers offered conceptual constructions, grand narratives postmodernists would say, to organise that universal scheme of things, but the one most interesting to this study must be Kant, who wrote *the* seminal book in the field of aesthetics: *Critique of Judgement* (1790) (Kant 1957). In it, Kant establishes beauty, and the aesthetic feeling it arouses, as entities separate from the theoretical – pure – and the practical logic. In other words, he established the difference between beauty, logic and moral.

The main characteristics of aesthetic judgements, according to Kant, are self-referentiality (subjectivity) and self-satisfaction (universality): I am satisfied with my own judgement about the beauty of something and believe it beyond correction (Townsend 1997); in other words, I do not need to look anything up to find out if something is beautiful (I possess all necessary knowledge), and telling me that I made a wrong judgement (what I thought was beautiful actually isn't) makes no sense. This is a concept of aesthetic judgements that still holds today, even if there have been quite a few modifications in matters of taste, and what is considered beautiful now may have been ugly at the time and vice versa.

But it is not only taste that has changed, Kant's conceptual construction has also seen modifications, even if the basics still form the conventional view on beauty and art. The characteristics of the aesthetic feeling, notably its immediacy and strength, have been the subject of much debate, and a number of philosophers have felt the need to launch their own inquires into the matter. At any rate, subjectivity and universality have remained the main traits of aesthetic judgements, a notion popularly summarised in the saying "beauty is in the eye of the beholder." Theories of beauty canons, which had quite some importance in the classical periods, have receded and the search for the right proportions is no longer a central project in any major aesthetic movement.

Nevertheless, if I have chosen "aesthetic ideals" as the title of this chapter I am not trying to suggest that that there are canons of beauty in programming, let alone a branch in a philosophy of programming aimed at establishing them. In this case, 'aesthetic ideals' is used in its vaguest – simplest – sense: as the desirable qualities that make software beautiful.

Now as much as I hope that the sentence "as the desirable qualities that make software beautiful" helped to explain what I mean by "aesthetic ideals", it is a bit problematic. It sounds as if I am going to offer some objective characteristics that make software beautiful. But this is impossible, as Frank Sibley points out (Sibley 2001), since aesthetic concepts, or concepts of taste, are "not, except negatively, governed by conditions at all" (:8). In other words, it is impossible to decide whether an object is beautiful just by enumerating a number of "non-aesthetic" features. On the contrary, we explain beauty with other aesthetic concepts, such as elegance or delicacy, which are just as irreducible to non-aesthetic qualities. Beauty can however, be explained with concepts that are *primarily* non-aesthetic, such as cleanness, simplicity or colourfulness, but only through a linguistic transformation. We are so used to this kind of transformations that we do not notice them. For instance, take colourful: there is nothing in the adjective colourful that would prevent a colourful painting from being ugly, so if I use 'colourful' to describe why a painting is beautiful, I do not just mean that 'it has a lot of different, and vivid, colours'. I mean that it has a lot of colours *and* that they are beautifully combined. When we use 'colourful' in an appreciative sense we change its literal meaning, adding an aesthetic qualifier. In order for that aesthetic 'colourfulness' to become a non-aesthetic term, I would have to explain, non-aesthetically, what 'beautifully combined' means. And this is what Sibley says is impossible.

This, Sibley continues, is not due to "an accidental poverty or lack or precision in language", neither is it "simply a question of extreme complexity" (:11); it arises from the non-conditional essence of the aesthetic judgement: we do not use any rules to decide whether something is beautiful or not. This, however, does not imply that it is impossible to discuss about the aesthetic value of objects. Only that each particular case must be considered on its own merit, and that we should give up the idea of reducing aesthetic judgements to a matter of enumeration of qualities.

The aesthetic concepts used by programmers, presented later on in this chapter, are all, apart from beauty and elegance, examples of primary non-aesthetic terms (cleanness, simplicity, tightness, etc.) used in an aesthetic sense. In some cases, the words used by programmers may confuse us into believing that they are not discussing the aesthetic but the public aspects of software. For instance, we shall see, how in some instances they may refer to a program's efficiency.

In some of these cases, the use of adjectives like 'efficient' is so obviously disconnected from a public perspective on code that there is no doubt that we are dealing with private aspects of software. In those cases, the context of use leaves little doubt as to the appreciative sense of the word (i.e. aesthetic (Wittgenstein and Barrett 1966)). In others, the concept 'efficiency' forms part of an instrumental belief, and the appreciative nature of a statement such as "this program is very efficient" is hidden behind a pseudo-instrumental claim (more about this in the next chapter, which considers the relationship between two fundamental concepts: functionality and beauty). Now, the adjective 'efficient' may of course also be used in an objective sense; for instance, when comparing the speed with which two programs carry out the same function. In some cases, describing a program as efficient may both be the result of a careful comparison and a proof of admiration. Luckily, it is often easy to see whether the adjective carries some appreciative sense or not. The following is a representative example of 'structured' being used to denote admiration (note the 'very'):

Older quality improvement techniques worthless? by **CactusCritter** (#2254711)
A lot of stuff from the 70's and earlier 80's seems to be unknown now. Was this due to proven lack of value or what?
IBM developed the Chief Programmer Teams which included lots of mentoring and non-author code reviews. There was proof of correctness via p-notation. (I never really mastered it, but I thought that was my fault.) There was truly structured code. Dahl, Dijkstra, and Hoare had "Structured Programming" published in 1972. I only saw Dijkstra mentioned in thie discussion. Pity.

A few last introductory words before turning to the aesthetic ideals themselves. Listening to the programmers' conversations about programming, we can distinguish a number of words which are frequently used to express their appreciation of code, the rest of this chapter is dedicated to their classification. As in the case with the coding styles in the previous chapter, and for the same reasons, I have no higher expectations as to its comprehensiveness. When I speak of 'programmers' it may sound as if I had a complete overview of the field, but this is not the case. I use 'programmers' instead of 'the programmers that took part in the Slashdot discussions and that otherwise form part of my empirical material' just for the sake of clarity. Once again, the point of the classification is not to be exhaustive but to illustrate how programmers refer to software in aesthetic terms, and to describe the private aspects of programming.

The case of coding styles was included to introduce the reader to the world of programming and its alternatives, even if the introduction finally only dealt with a tiny bit of the software development process. It was perhaps not as much focused on the aesthetic feeling as on the possibilities of personal expression, and, let me insist, it only dealt with the limited possibilities offered by transparent coding (alternatives that were transparent to the compiler). The non-programmer reader that has read the previous chapter may perhaps imagine the enormous possibilities to personal choice offered to the programmer throughout the whole development process: from specifications requirement, design, coding, testing, debugging, installation, etc.

Any classification of the aesthetic terms used by programmers must deal with the fact that as much as they are, have to be, extremely exact with the use of commands in programming languages, they can be as careless as anyone else with words like 'elegant', 'clean', 'beautiful' etc. Arguably, part of the reason for this carelessness lies in these two circumstances: firstly, in all likelihood, they have neither the time nor the interest to weigh every word they use when they are publishing opinions on the internet or being interviewed by researchers; secondly, and most importantly, aesthetic opinions are seldom well defined: expressing the feeling that a given object conveys is an art in itself, and normal programmers are not much better trained in that than the average citizen. But, once again, we do not seek the chimera of an objective, comprehensive classification, but a description of the programmers' relationship to software.

ELEGANT AND BEAUTIFUL

These two are the adjectives that programmers use most often for expressing their aesthetic judgements about programs. I have chosen to present them together because these are practically the only primary aesthetic words used by programmers to signal aesthetic feeling. The rest of the terms presented here are, in Sibley's terminology, non-aesthetic and are often accompanied by 'elegant', 'beautiful' or some other word that indicates their aesthetic sense (such as '*truly* structured' in the previous example).

'Beauty' and 'elegance' – the latter perhaps even more so – are often used as the paramount characteristic of software, as C. A. R. Hoare did in his 1980 Turing Award lecture: "I have regarded it as the highest goal of programming language design to enable good ideas to be elegantly expressed" (Hoare 1981).

However, simply saying that a program "is beautiful" does not reveal much about it, and what we tend to find in the programmers discussions are combinations of the sort:

Simplicity and elegance
Highly commented and very elegant
Beautiful and elegant
Clever or elegant
Beautiful or elegant
Elegant & concise
Creative and elegant
Sublime elegance
Elegant simple
Wonderfully elegant
Elegant and masterful
Quiet elegance
Elegant, masterful, excellent and beautiful
Pure elegance
Purity and elegance
Incredibly beautiful well structured
Elegant or "pretty"
Great, beautiful, engaging, pleasing
Elegant efficient and really readable
Functionally beautiful
Simpler, more elegant, and more readable
Bugfree and beautiful
Concise, elegant and above all logical
Beautiful, maintainable
Beautiful robust
Beautiful and usable
Pretty or beautiful
beautifully simple

All those combinations are taken from Slashdot messages, in which often the participants claim something about beauty in programming and then go on to explain what they mean. The following is a representative example:

Re: Beauty for beauty's sake makes crappy software by **Anonymous Coward** (#2254673)
I always had the experience that beautiful code is superior to ugly code. That depends on the definition of beauty in that case.My definition of beauty comes from math and physics which means that a shorter more compact formula is more beautifull than a formula that has to take into account many special issues. What is nicer to look at? Code where you have several if() clauses for several sepcial cases, or code where the if() clauses can be omitted because the code is designed and structured in a way that includes all these special cases without the need of having to consider special cases at all.

Even if they are the only primary aesthetic terms in the Slashdot discussions, the use of these two words is not what proves the existence of aesthetic aspects in programming. This existence does not depend on the kind of words used, but on the kind of attitudes towards code that programmers have. Furthermore, if these words were the only 'aesthetic' about programming, there would be little point in writing a thesis about it: it would be enough to mention it en-passant. But programmers respond to software, or relate to it, in aesthetic ways, that is, they do things with it that one only does with aesthetically pleasing objects. Remember the introduction to Slashdot's discussion *Where do I find beautiful code?* and consider the following entry to that discussion:

Re:TCL or AOLServer by **velouria** (#483088)
I downloaded the Tk source a couple of years ago to make a small modification and ended up printing out and reading the whole thing for the pure pleasure of it. It shows that C can be very easy (enjoyable even) to follow. The code's modularity and reuseability is miles better than most of the "Object Oriented" code I've seen too.

The existence of this kind of phenomena – like *velouria* reading the Tk source code "just for the pleasure of it" – are the real manifestations of the aesthetic aspects of programming. This does not mean that all programmers relate to the beauty of software in the same way. Not all programmers, for instance, read code just for the pleasure of it, but more importantly, not all programmers say that beautiful software is something desirable. This difference of opinions is an interesting subject in itself, and the next chapter is dedicated to it, but for the moment I shall avoid it, limiting this one to an exposition of the terms used to describe the beauty of software.

There are many other adjectives that can be used in the same way as 'elegant' and 'beautiful' but that turn up much less frequently: nice, great, marvellous, enjoyable, fine, excellent and other similar ones. Naturally, there are also quite a number of adjectives that allow the programmers to express a general feeling of dislike. The most often used is ‘ugly’ but there is a whole family of negative evaluators: butt-ugly, horrid, horrible, terrible hell, shit (also as “S#!%”), sucks, displeasing, poor, hideous, crap and more. Finally, ‘pretty’ should perhaps also be included in this group of primarily aesthetic adjectives. ‘Pretty’ is mostly used as ‘nice’ but sometimes we can see it showing nor pleasure or displeasure but rather despise (as I said, not all programmers think software should be called ‘beautiful’):

Beauty.. by **gnireenigne** (#483225)
.. is in the eye of the beholder - this aphorism works in this case too. But anyone who has had a chance to look at M$ MFC or Native Winblows code would have to agree that you would have to be blind to find beauty in any of that kludge[25]. But code is not s'possed to be pretty. It's gotta work. That's what it's there for.
Code On.

Summarising, ‘beautiful’ and ‘elegant’ are clear signs of aesthetic appreciation, often used to indicate the tone of the message. But since they do not refer to any characteristic of software, they are not really aesthetic *ideals*, they do not describe wherein the beauty of the program resides. The rest of the terms presented below are, on the other hand, proper aesthetic ideals: with their help, it might be possible to distinguish between different ‘schools’ of programming… even if programmers themselves do not seem particularly interested in that notion.

‘Clean’ is used quite often among programmers and, as is customary with words that are used often, it can mean several things. The most frequent meanings of “clean code” are ‘readable code’ (white spaces and empty lines that properly indicate of the structure of the program, well implemented indentation, suitable names, etc.) and ‘lucid design’ (clearly distinguished functions, forms of access to databases and other programs, etc.). Notice that in the following examples the word ‘code’ is not always limited to the actual code, but used metonymically for the whole program:

coldsync is a good example of The Right Way (tm) by **mcoletti** (#482985)
coldsync, a Palm Pilot synching utility, has the cleanest code I've ever seen. The design is well thought out. The source is extremely well commented. It also has a lot of documentation, which has the bonus of being informative, comprehensive, and otherwise very well written. It also has some good "meta-level" documentation; e.g., the top-level "HACKING" file, which gives some basic pointers for those wishing to, well, hack the hell out of the thing.

Re:FreeBSD et al. by **arnald** (#483263)
Better still, take a look at NetBSD - much cleaner. :-)

not many by **cabbey** (#483023)
In many of the larger projects though you can ocasionally find bits and pieces of pure poetry in code. There's an example in the Linux kernel, I forget exactly where - maybe in the vmm, where someone took the time to fully digest a rather hairy function and they totally rewrote it without changing the inputs, output, or side-effects in a small clean block of code.

So from well thought out design and “extremely well commented” source [code] to just “much cleaner”, to “small clean block of code”. One of the few things that is common to all uses of the adjective ‘clean’ is that it refers to something positive. It seems that the word ‘clean’ in "clean code", "clean programming practices", "clean non-local error handling", "clean design", "clean style", "clean logic" and "clean software" refers in all examples to something good, but perhaps not necessarily to the same thing. Let us examine a few of those uses.

Referring to design, 'clean' may mark the absence of "dirty tricks", which are programming stratagems "to squeeze every bit (pun intended) of performance out of [programs]". The idea of using tricks was explicitly mentioned by the legendary Edsger W. Dijkstra in his Turing Award lecture:

> The competent programmer is fully aware of the strictly limited sized of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. (Dijkstra 1972)

He calls them "clever" tricks, I'd swear it's an irony, and he definitely does not like them. He does not like either the programming languages that stimulate the use of tricks, as he explains in the lines that follow the previous quote:

> In the case of a well-known conversational programming language I have been told from various sides that as soon as a programming community is equipped with a terminal for it, a specific phenomenon occurs that even has a well-established name: it is called "the one~liners". It takes one of two different forms: one programmer places a one-line program on the desk of another and either he proudly tells what it does and adds the question "Can you code this in less symbols?" - as if this were of any conceptual relevance! - or he just asks "Guess what it does!". From this observation we must conclude that this language as a tool is an open invitation for clever tricks; and while exactly this may be the explanation for some of its appeal, viz. to those who like to show how clever they are, I am sorry, but I must regard this as one of the most damning things that can be said about a programming language.

This preference for cleanness over clever tricks can be found all over the, so to speak, *official* (teachers, venerable programmers, etc.) view of programming. Serious programmers hold it almost a moral maxim, that frivolous stuff like clever tricks should be avoided. However, there appears to be little sign of that frivolous stuff disappearing. Easter eggs (little surprises hidden in the code), backdoors (unofficial ways to enter a system), obscure code contests (we have seen a few examples of the entries to these), one-liners, hacks (in the sense of ingenious programming stunts) and other frolicsome behaviours are as thriving as ever. They are indeed an important part of the private aspects of programming, but will not be dealt with in this study.

Cleanness relates, in this sense, to a programming approach that praises transparency of thought over cleverness. One of the ways to make sure one's thoughts are transparent is to think in a classical manner, classical in the sense of following some basic, classic, rules of programming. A few postings make allusion to this notion:

OpenBSD by **kinger** (#483093)
If I recall the OpenBSD[26] guys don't necessarily audit the code for security, they focus on good, clean, correct programming practices. Security then naturally falls into place.

Andrew Tannenbaum's original Minix by **AReilly** (#483025)
[...] It's nice, clean, traditional C style, too.

IP Filter by **bbeausej** (#483340)
That's one of the reason why I really like digging into Darren Reed's code of IP Filter, I think it is ressourceful to see good code like this, that implement standards correctly and cleanly, and that use the features of the language without abusing them.

The expressions "good, clean, correct programming practices", "nice, clean, traditional style" and "standards [implemented] correctly and cleanly" all insinuate an idea that could perhaps summarise the essence of cleanness in programming, if there is one: it is about doing things properly, seriously. This is all very vague, but not so vague that programmers would not know when they are on the wrong side of the boundary:

Re:Beautiful software by **quartz** (#2253179)
[...] I know I should strive to make my code "readable", but the irresistible urge to type: join(" ", map { ucfirst } split(/ /, shift)); just to correct a spelling error in the database on-the-fly is simply overwhelming. I just can't help myself. :)

(S)he knows that one-liners are not exactly best practice, but feels an "irresistible urge" to create them. As much as 'clean' is clearly something positive, it also might be unfashionable. And it would be foolish to imagine that programmers, unlike other human beings, do not like to show off every now and then. But this is the frivolous stuff trying to find its way back to our text...

An expression derived from 'clean' is 'to clean up', which is also rather popular amongst programmers. 'Clean up' is what they do to programs that are deficient and need rework. Exactly what it is that might need rework is difficult to explain without getting into technical details, let us simply say that some bits of the program may have been developed in a rush, and the programmers may feel that their work is not ready to be shown, it is too sketchy.

Re: NeoMail by **happystink** (#483270)
wow, they must have really cleaned it up from a few months ago then, cause back then it had a security flaw so super-basic that it showed that they didn't really understand a single thing about web security

Don't scoff by **fractalus** (#2253006)
[...] So earlier this year I began pushing for a major project to refactor and clean up this code. Already the initial stages of this project (implemented by several programmers on our staff) have yielded huge gains in how quickly we can develop stuff. As we continue to clean and simplify, these gains increase.

Re:*ACK* VBScript!! by **EWillieL** (#2254034)
I feel your pain -- BUT -- Visual Interdev is NOT VB! VBScript is indeed an abomination against God and Nature, but VB is maligned only because its early versions were implemented well enough to allow idiots to use it and thereby proliferate mounds of horrific spaghetti code (written without a plan, to be sure).I do often wish the idiots had been left out of the VB game, since I now have to go clean up after them (like on the project I'm working right now -- yeesh!).

Simple is an adjective that is used as vaguely by programmers as by anyone else. Also in programming, it has both a positive and a negative connotation. The negative one has to do with descriptions of problems: if they are simple they do not present the programmer with a real challenge, they are not of interest. We can observe this use quite a few times in the Software Aesthetics discussion. Remember that Connell's article (Connell 2001), the one that triggered the discussion, was based on a comparison between civil engineering (building bridges and houses, particularly) and programming. Well, some participants thought the task of civil engineers was much simpler than that of the programmers:

Half right. by **antis0c** (#2252976)
It's half right to say we should engineer software like we engineer physical aspects of our lives such as bridges, houses, skyrises, etc.
However.. Bridges, Houses, Skyrises, all are slow moving, slow changed things. House "technology" doesn't suddenly double every 18 months. Otherwise we'd be like the Jetsons with houses into the sky, and talking dogs. Bridges are engineered to last for a very long time, because they do a simple, easy function. They provide land where there is none to travel on.
Software and Hardware however does. If you talk to a lot of software engineers, professors, etc. They'll all usually say not to worry about performance, next years computers will run it twice as fast anyway. This is very true at Microsoft. And its partially correct. Why bother spending a ton of time trying to make something work fast, when during the time you took to make it work fast, chips have doubled in speed?

This vision of civil engineering as a simple activity is not widespread, some programmers complained about the lack of respect shown for a profession about which most participants did not know anything. There is, on the other hand, a rather extended sentiment that programming is extremely complex, mainly because of the intensive changes in the technology, the lack of clear specifications when designing a program, the shifting of priorities halfway through the projects and so on and so forth (in this, they share Brooks' notion of software being something special, and of software development projects of requiring specially dedicated studies). In many cases, this complexity is used to excuse the poor quality of the software that, practically everyone admits, is being produced.

However, this is not what I would call an aesthetic use of the word 'simple'. It is in its positive use that 'simple' expresses someone's appreciation of something:

Python by **AMK** (#482996)
Try looking at the source for the Python interpreter (the C version; I know little about JPython). Like any sizable program it has its messy bits, but overall the organization is quite clear, the implementation leans toward the simple and straightforward instead of the weirdly optimized and obscure, and it's one of the more pleasant medium-sized programs I've worked on.

In this appreciative use, 'simplicity' means very much the same as it means in every other human activity: a sort of clear straightforwardness, something that could perhaps be illustrated by classicist ideals, as opposed to baroque ones. Simplicity in programming can be achieved in as many different ways as it can in the realm of, for instance, architecture. For example, a deeper level of abstraction, that allows the programmer to avoid messy constructions to take care of special cases, or the right statement, instead of an unnecessarily long combination, can both be ways to simplify a program:

Re: OpenSource is Beautiful by **Alan** (#482982)
[...] For example, I had a great routine[27] for getting system mem/cpu info from /proc/*. Browsing through the wmsysmon code I found that (to my surprise) there is a sysinfo() call that can give me that exact information... suddenly my code is made simpler, more elegant, and more readable, as well as more bugfree thanks to one line in an obscure little applet (well, maybe not *that* obscure :).

Programmers seem to take pleasure in finding those commands that simplify their programs (as in making the design more straightforward and diminishing the number of operations/commands)[28]. You must keep in mind that a programming language like C comes with a large number of libraries, i.e., extensions that contain ready-made functions (commands), which allows for a great level of redundancy and alternative writing of the programs. At any rate, finding the right "call" can at times feel like discovering a hidden treasure (apart from the fact that it shows how well-read you are).

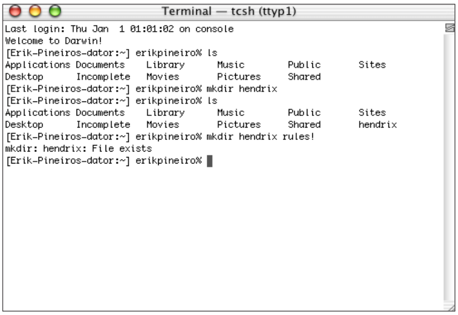
Simplicity in programming is sometimes connected to asceticism. Some programmers use 'simple' as a way of expressing their admiration for a bare-boned program, that includes no unnecessary ornaments. This is a very similar aesthetic position to the one we saw when going through the coding styles (minimalism). Quite a few programmers extend their admiration for simple code to the gui (graphic user interface) and prefer applications where the user has to write down every command instead of clicking on nice buttons placed upon colours schemes and so on:

And, by extension, xdvi.... by **Booker** (#483015)
[...] When I was using TeX, I was using xdvi as well - and that also really impressed me. Simple interface, SUPER snappy response on the gui - no animations, alpha blending, or themes...
Click on that magnify button and zoom the window around... ahh..... does anybody write code like TeX any more?

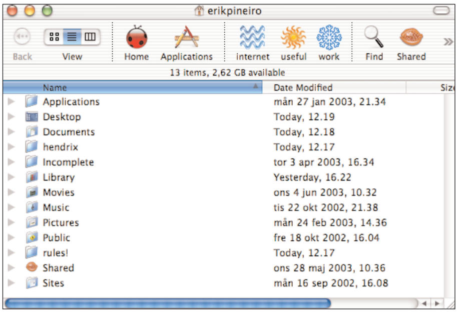
Re:flawed analogy by **Anonymous Coward** (#2257156)
[...] I miss the old command-line mail system days from MIT when I knew what e-mail I was getting and I was in control of it - not some overblown GUI contraption...

This aesthetic ideal can be compared to functionalism, which in the case of programming is sometimes accompanied by a somewhat reactionary air: some of these programmers express their dislike for the way programming is evolving, from the old austere command-based operative systems and programming environments to the new user friendly and colourful ones.

A Prime Example by **docstrange** (#2252864)
Is it just me or is computing going backards. It used to take 2 seconds to boot into DOS, yet it takes 20 seconds or so to boot into Windows 2000. We have various gui based medical management software that we manage at my place of employment, but the old dos versions were far more efficient. Call me a hermit, but I think that the "user friendly, gui=productivity myth" needs to die. Visual basic should not be used for ANY production applications. Especially in mission critical system. That aside, I would like to thank all of the crappy programmers, for providing me with job security. As long as your stuff breaks, my future looks bright.



```
Last login: Thu Jan  1 01:01:02 on console
Welcome to Darwin!
[Erik-Pineiros-dator:~] erikpineiro% ls
Applications Documents    Library      Music        Public       Sites
Desktop      Incomplete   Movies       Pictures     Shared
[Erik-Pineiros-dator:~] erikpineiro% mkdir hendrix
[Erik-Pineiros-dator:~] erikpineiro% ls
Applications Documents    Library      Music        Public       Sites
Desktop      Incomplete   Movies       Pictures     Shared       hendrix
[Erik-Pineiros-dator:~] erikpineiro% mkdir hendrix rules!
mkdir: hendrix: File exists
[Erik-Pineiros-dator:~] erikpineiro%
```



Name	Date Modified	Siz
Applications	mån 27 jan 2003, 21.34	
Desktop	Today, 12.19	
Documents	Today, 12.18	
hendrix	Today, 12.17	
Incomplete	tor 3 apr 2003, 16.34	
Library	Yesterday, 16.22	
Movies	ons 4 jun 2003, 10.32	
Music	tis 22 okt 2002, 21.38	
Pictures	mån 24 feb 2003, 14.36	
Public	fre 18 okt 2002, 16.04	
rules!	Today, 12.17	
Shared	ons 28 maj 2003, 10.36	
Sites	mån 16 sep 2002, 16.08	

FIGURE, Two ways of accessing your files, *docstrange* prefers the one on the top

The opposite of ‘pleasingly simple’ is not so much ‘disgustingly complex’ as ‘messy’. With the word ‘complex’ programmers frequently describe programming projects and their circumstances. In this sense it is used to accentuate the difficulty inherent to programming and, by way of this, the great achievement it is to be able to write beautiful software (“The complexity of large software projects is unprecedented. I've often explained software to my (non-technical) friends as similar to building a car out of watch parts. And those are only for the smallish projects I've worked on.”)

The concept of utter messiness is perhaps most vividly expressed with the term ‘spaghetti code’. Clearly, spaghetti code is not simple; the problem is that neither is it clean, which may serve as a reminder of the fragility of any classification of aesthetic ideals. Another concept that is opposed to ‘simple’ is ‘over-designed’, used to designate software that includes too many features and can break in too many places. The richness of the concept of simplicity in programming can be illustrated with the following Slashdot message:

Complexity is in YOUR own perspective only by **Taco Cowboy** (#2254706)
Simple or complex.Straight forward or beating around the bush.Keep It Strictly Simple, or make it as messy as possible.It's ALL up to you. I write software for a living.I can produce speghetti code, if I want to. I can make the software as complex and as difficult to use as possible, if I choose to.But why should I?Simple things come easier for me. I am a simple guy, living a simple life. Sometimes I make a mess out of myself, but most of the time, I ain't that messy.Therefore, before I start to code ANYTHING, I think, I plan, I scheme. I look at other people's code, I steal ideas from them, I mix and match things that I like, and I incorporate whichever is useful in the program I write.When all is said and done, what I aim for is to produce something that is simple, and yet powerful.Clumsy software not only hurts the users, it also hurts the authors too !

'Tight' is an adjective that, in everyday language, does not necessarily mean 'pleasurable'. However, whenever I have seen it in programming environments, it was always used in an appreciative sense. It can be used in two different general directions, one related to design (or programming in general) and another one to coding. In the first use it indicates that the program has been carefully designed and that all its pieces fit tightly (they use the memory in an efficient manner, they use each other's functions perfectly, there are no possibilities for deadlocks or ambiguous priorities, etc). In the second use, it indicates that the code has been written in a minimalist (see the *coding styles* chapter) style.

Writing tight programs may imply a degree of unreadability since, figuratively speaking, there is less space between each one of its parts. In fact, the most extreme examples of tightness are the entries to the obfuscated contests. Both the design and the coding of most of those examples are exceedingly compact, and they are not without admirers as we saw in the previous chapter. More everyday exponents of tight code seem to exist a little bit everywhere:

What is "elegant" code anyway? by **foobarista** (#2253952)
Is it:1. Massively documented, heavily object oriented code with lots of reuse, designed for ease of enhancement and maintenance, etc?2. Tight, fast-executing code which, while it may be difficult to figure out at first glance, is powerful, resource-sparing, and generally cool algorithmically? Of course it is well documented, designed carefully, etc.3. Other standards of elegance?There are jobs for which (1) is the definition, and others for which (2) is a good definition - although my own bias leans toward (2). Note that systems built with (1) as the definition tend to be resource inefficient, but in many applications, this doesn't matter. Are you writing code which glues together a bunch of legacy db's and other apps or are you coding an OS or filesystem? To borrow a phrase from architecture: form follows function.

As *foobarista* suggests, 'tight' in programming is related to "fast-executing", "resource-sparing" and "powerful", all of which are obviously much better than their opposites. The tradition of writing tight code can perhaps be traced back to the conditions that the first computers put on the programmers. The amount of bytes of memory they could work with was almost infinitely smaller than what is available today in any mainstream PC and programs had to consume as little of it as possible. Also, computers were much slower than they are today and calculations and access to data took years – well, tenths of seconds, or even seconds. All these circumstances forced programmers to minimise execution times and database searches and everything else, turning programming into the art of compressing – and creating the Y2K concept. Stephen Levy's book about hackers (Levy 1984) tells the story of some of their achievements, the most admired of which were those that made the computer do things that seemed unfeasible. Today not everyone is in favour of this sort of programming style but it lives on and has enough force to raise up some disputes:

Re: The Story Of Mel (or: Ugly Code) by **noahm** (#483000)
This classic hacker legend came up in a software design class while at Northeastern. It was used as an example of terrible coding practice. Sure, Mel was an artist, and an incredibly skilled programmer, but his code was completely unreadable. Unless you're one of those people who follows the "if I write unreadable code then they can never replace me" school of thought for job security then you should definitely not look up to Mel.

Re: The Story Of Mel (or: Ugly Code) by **Mark Imbriaco** (#483198)
What you're failing to consider is that Mel was coding for a far different era of computers. Not only did he do things the way he did because it was *cool*, he did it because computers weren't really all that fast and you had to play dirty tricks in order to squeeze every bit (pun intended) of performance out of them that you could.
-Mark

Re: The Story Of Mel (or: Ugly Code) by **noahm** (#483001)
I get the point, I just strongly disagree with it. Code needs to be readable in order to be useful. I aknowledged Mel's skill and creativity in my original post, but I stick to my point: he was a poor engineer. I'm sure you've heard the quote "If builders built buildings the way programmers write programs, the first woodpecker to come along would destroy civilization." That describes Mel's coding style perfectly. Make the tiniest change to the system it's running on and the whole thing falls apart, requiring a complete rewrite. That's rediculous.
I don't know where in my post you find any reference to debuggers that hold your hand or some kind of context sensitive help system or whatever. I sure never mentioned that. I'm just saying that Mel was a bad programmer because his code was unnecessarily obfuscated and completely inflexible. It has nothing to do with what tools he did or did not use. Obfuscated, inflexible code can be written with the latest GUI IDE. It doesn't take an old timer with a hex editor, and any hack value it has is quickly lost when you're the one tasked with figuring out why the code won't work after upgrading the system.

Re: The Story Of Mel (or: Ugly Code) NOT by **Anonymous Coward** (#482980)
god damn it!
Ok, yea, right there in the hex, you should put a few lines like

```
# function for generating the card to be delt to the
user from the computer.
```
[29]

Ok, so with compiler coding (post-Mel), you could do this, and that useless ascii would total maybe half a kilobyte, maybe one. In olden days that was a gabillion dollars!!
Arogent people who bash those before them with their 20/20 hind sight piss the shit out of me.
Mel wrote the fastest and most optimal program possible, hands down. That's not a negotiable fact. He went the extra mile(s) to get get it working the fastest, at which point, it became a work of art. Since Mel, things have changed, and now greener pa$ture$ rule the world. That wasn't the case back then, when conserving resources was everything. Why do you think we had y2k? Because it was that damn important to save resources. That was the real cost.
Anyone who calls Mel's code Spaghetti code should go to hell. Mel was an artist. That's like saying that frank lloyd wright's blueprints where scribble, because his buildings couldn't go through a tornado. Mel was an architect in a world of engineers. To quote RENT, "take him for what he is". Damnit, that really really pissed me off.

Other examples of tight code are what Dijkstra called the 'one-liners'. Even if he did not have much good to say about them, they seem to attract some programmers, remember for instance how *quartz* expressed his or hers "irresistible urge" to type them. According to *quartz*, that construction (*join(" ", map { ucfirst } split(/ /, shift))*) would require a "10+ line" code "in C++ (or, god forbid, Java)", something that a) makes it pretty tight and b) shows that *quartz* has some mastering of Perl (the language in which the line is written). This way of making one's code tight might remind you of one of the possible meanings of 'simple' in program formulation: finding the right command. It would seem *quartz* has found the right combination of functions that translate 10+ lines of code into a simple one-liner. Now, whether one liners are a question of tightness, simplicity, proficiency, baroqueness or irresponsibility is an open question.

The fact that writing one liners, and tight code in general (clever tricks), requires skill and cunning is perhaps the central element of tightness as an aesthetic ideal in programming: achieving it is seen, by some programmers, as a proof of mastery. There are many voices (Dijkstra's and Knuth's amongst others) that advocate for a more self-explaining and readable code but there are also quite a few that support tighter coding. On the other hand, I would not like to imply that all forms of tight coding are against Dijkstra's and Knuth's ideals. I cannot imagine that they would be against an optimal use of the processor resources or for writing many more lines of code than needed. As a more concrete example of a form of tightness as one of Knuth's ideals is his use of low level languages in his books. One of the ways of tightening up one's program is to write in assembler, or at least in a low-level language (languages that are near the machine code) since that gives the programmers a more exact control of the processor's actions. All the examples in Knuth's seminal *The Art of Programming* (Knuth 1997) are written in a low level language, to allow the tightest possible control over the processor.

Code written in a high level language (what most programmers use) must be compiled into a series of operations that the processor can carry out (see the beginning of the *coding styles* chapter). The compilation is usually carried out opaquely, in the sense that programmers cannot see the exact operations into which their code has been translated. This inserts some sort of slackness in the program since one is not sure of exactly how the compiler will implement one's high level code. Some programmers seem to quite dislike this and therefore go through the pain of writing their programs in assembler - or at least the central part of their programs. This is the reason why Knuth chose a low level language for his examples in *The Art of Programming* and also what *tarsi*210 refers to:

Compilers depriving us of beauty? by **tarsi210** (#483113)
From the: Borland-has-nothing-on-Van-Gogh dept.
It seems to me that the bulk of "beautiful code" which is produced today is that which is still written in an editor at a low level to be compiled on the fly or to do low-level, system-admin type of stuffs. (IE: Perl, shell, etc.)
Since making it through the scads of OOP methodologies and theories in college, I have seen little to nothing in the way of high-level (C++, Java, Smalltalk, PowerScript) languages that come even close to the pure genius of a nicely written Perl function.
Is it just me, or have our fancy compilers and IDEs and methodologies removed from us that intricate, complicated, horribly frustrating task of doing low-level, compact work that now we rarely see something of beauty? When was the last time you read some good assembly that just made you smile because it was so damn clever?
Swiss watches still sell well because they have something that the $5 digital watch from Wal-mart doesn't. Intricacy of the parts, the beauty of the syncronization of movement, the pure elegance in physics.

Coding in assembler (remember that each sort of processor has its own assembler language) might very well be necessary for programs with tough performance requirements, or at least for parts of it, but it has nowadays lost the significance it had at the times of strict hardware limitations. However, it persists amongst programmers as a sort of subculture, very much like the Atari or the Commodore (really ancient computers with low graphic possibilities and not much processor power) gaming clubs.

I suppose the opposite of 'tight' would be a program designed with complete disregard to the number of lines of code, whether there are any routines that can be reduced to a command, the amount of memory it uses, the number of time consuming accesses to databases it carries out, etc. I have been unable to find any adjectives that would describe exactly this sort of code. It can be called ugly or, perhaps more often, sloppy, but both of these are used to describe other sorts of code as well (such as ill-structured and un-clean). The fact that the opposites of so many adjectives seem to be common confirms the idea that the ideals that I am listing here are in no way clearly separated.

Anyhow, *gojomo*, one of the participants, describes ugly programs as those with the following characteristics: "complexity, poor modularity, leaky interfaces, and sacrific[ed] design and comments in the name of rapid deployment". 'Poor modularity' and 'leaky interfaces' are quite the opposite of tight: a program has poor modularity and leaky interfaces when its parts do not fit well in the overall structure or amongst themselves. But the interesting thing is that *gojomo's* entry is actually a apology for these characteristics:

Worse is Better by **gojomo** (#2254683)
Charles Connell's essay presents appealing ideas; it'd be nice to think software could be "more aesthetic". However, the truth is that Worse Is Better [dreamsongs.com] -- as Richard P. Gabriel will often argue

(and other times refute). Exactly those things which make software "ugly" -- complexity, poor modularity, leaky interfaces, and sacrificing design and comments in the name of rapid deployment -- can also help make it more successful, commercially and socially, in the long run. Exactly those things which make software "beautiful" -- such as Connell's qualities of "appropriate form", "minimality", "component singularity", "functional locality", "simplicity", and even "readability" -- can in fact make software fail, as "great programmers" spend a bunch of time making "elegant solutions" that never catch fire, because they lack the immediacy and approachability of more haphazard solutions.While this idea may sound like an excuse to avoid doing up-front thought, it's actually a hard-earned lesson that what aesthetically appeals to good, well-intentioned programmers may in fact involve all the "wrong" tradeoffs.Read all the stuff on this topic at Gabriel's Worse Is Better pages [dreamsongs.com], then revisit Connell's aesthetics peice, and Connell may seem downright naive to you.

gojomo suggests reading Gabriel, and Gabriel's notions on how to write useful programs are opposed to the ideal of tightness. His ideal, habitability, is not opposed to leaky interfaces and poor modularity, since this may be exactly what makes the program truly useful, as *gojomo* explains. Those imperfections allow programmers to work with more ease when maintaining and improving existent programs. Truly tight programs can be a nightmare to maintain because any small change made on them or its environment can have unexpected consequences; as *noahm* put it: “make the tiniest change to the system it's running on and the whole thing falls apart, requiring a complete rewrite”[30].

‘Tight’ has a number of cousins, other adjectives that mean very much the same. Some of them are so closely related that perhaps they should be called sisters. For instance, I personally cannot find any significant difference between the uses of ‘efficient’ and ‘tight’. In fact, I could have explained ‘tight’ as ‘efficient in terms of use of memory space, lines of code, modules and processor time’. It appears in quite a few entries (“I've been pained watching my coworkers code in the past, replaced existing routines with amazingly smaller and more efficient pieces"[31]) and it carries, as far as I can see, very much the same meaning as 'tight'.

'Small', which appeared in the discussion about assembler ("small is beautiful") appears also in a few postings and its meaning is rather straightforward: either few lines or little space in memory.

Another adjective that I would say is closely related to 'tight' is 'fast'. In the world of programming there are at least two different kinds of speed: that of the program when executing and that of the programmer when writing a program. Only the first one is related to the tightness of a program, in a rather straightforward manner: one of the results of writing tight code is its rapid execution, or perhaps it is more correct to say that tight code is code that runs fast.

The second one is less connected to the beauty of code, but appears nevertheless often in the discussions about software aesthetics. There it not only refers to the creation of the program but also to its maintenance. There is much admiration for the skills of those programmers who can write beautiful code quickly and there is also some complaint about managers (and other programmers) that are not able to understand the deed of writing good code on the first try:

Making it "beautiful" IS making it work. by **Ungrounded Lightning** (#2253315)
[...] For instance: I habitually use these principles in my coding, and debug as I go. My work was once characterized as "... takes three times as long as anyone else, but it usually works."Bullshit.The techniques made my work so blazingly fast that I was able to deliver a complete, debugged, essentially error-free component in about three times what it took the other programmers to get to their first clean compile, or to do a debugging iteration. And by "essentially error-free" I mean that in over two years of work at one site, with thousands of lines of code, I had one error detected by someone else, in a preliminary internal release, before I had corrected it.But the result was that my time-to-

completion was measured against everybody else's time-to-do-a-debugging-iteration. And the administration discounted my advice in favor of that of the "most productive" - read least careful - member of the team. And the bulk of the project iterated until the sponsor turned off the money [...]

Then there is ‘optimised’, another adjective that I see is directly connected to the idea of ‘tight’ code. There are two slightly different uses of the word: a) after writing a program, programmers can return to it and rearrange bits that they believe could be better done (thereby optimising it); and b) from the beginning a program can be composed and formulated so that it is optimised in a particular sense, say, fast execution. Optimising a program for something, whether it is done from the beginning or once the program runs, is in many aspects the same thing as writing tight code. On one hand, being capable of really optimising a program is a proof of talent; on the other, an optimised program might be incomprehensible to anyone else than the author or, even worse, the design that results from an optimising approach may prove rigid and impractical (given that conditions for the project will probably change). The early hackers seem to have been masters of optimisation, according to Levy (Levy 1984) they even had their own verb: to bum (an instruction), which meant to write the same program with an instruction less. For them, being able to bum a program was a proof not only of mastery but also of belonging (the chapter on *beauty and functionality in programming* – section *true-geek attitude* – contains an illustrative quote on the subject).

But as much as optimisation in some cases allow programmers to achieve impressing results (and this is particularly true for the resource-poor early computers), it also makes the program quite unreadable and, consequently, unmaintainable This was not a problem for

Levy's hackers, who were devoted to computers and could find all the time in the world to decipher the programs and who, besides, could use this difficulty as a means for separating the true initiates from the novices; but it may be one for less extreme programmers. Examples from this other view of optimisation can be found in the online discussions, participants *Procrasti* and *beej* present a negative view on optimisation (and it is perhaps not so surprising that Knuth, founder of Literate Programming, is not much for it either):

Java is inefficient by **Procrasti** (#2253348)
[...] To sum up, Java is generally a more elegent language than C++, this leads to code with quicker times to market, less bugs and less cost in support and maintenance - efficiency isn't everything, afterall, "premature optimisation is the root of all evil" -- Donald Knuth, and how much more premature can you get than in choosing the implementation language?

Simpler isn't always faster by **beej** (#2253341)
Simple programs [...] run faster (because there are fewer machine instructions)
This can be true, but certainly isn't often the case. To optimize code for speed often involves contortions that do not clarify the code or make it simpler or easier to read.
The real trick is making it fast and readable. :-)

But they are far from representing all programmers. Readable or not, an optimised code is still viewed by some (perhaps many) as a sign of talent. *tjb* tells us how he spiced up his CV with a "insanely optimized" program, in order to impress the employer. According to the story, that seemed to help when looking for a job. *tjb's* message is part of a thread started by someone who blamed ugly software on the poor skills of programmers, hence the subject "Hiring better programmers would help." Someone else replies that despite being a good programmer he or she has difficulties to find employment, lacking an official degree. To which *tjb* answers:

Re: Hiring Better programmers would help by **tjb** (#2253920)
Dude, I don't have a degree either, but I got hired at a modem-chip company almost exclusively because I did send them code. Not that they asked for it, but I decided that I'd improve my chances by sending in some code.Along with my resume, I sent a 320x240 space-invaders program I wrote in straight 80x86 assembly. It wasn't a large, professional style project or anything (only about 750 lines of code, not counting art) but it demonstrated that I knew my shit. I program DSPs and small embedded devices now, and I have my insanely optimized little game to thank for it :) [...]

We can include in the same positive view of optimisation those who refuse to work with programming languages that do not allow hand-optimisation ("Damnit, I want a programming language that gives me access to the freeking carry flag! =). I've done math routines a lot, and the code is literally 10x faster when you can optimize it by hand in assembly. I love assembly for small things that you want speed for."[32])

Finally, there is the verb 'to squeeze', that is an expressive way to indicate the tightness of a program (notice how "most people didn't think it was possible" to do what Future Crew, Orange and other similar groups did, one of the characteristics of an awe-inspiring hack):

Doesn't anyone remember the 'scene' by **codetalker** (#483296)
Remember back in the days of Second Reality by Future Crew and groups like Orange et all.
If you can find the source to a demo, you can witness some truely awe inspiring code. www.scene.org These people squeezed every last clock cycle out of those old 386's. Most people didn't think it was possible. I know I didn't back then.

CONSISTENT

There are many different possible ways of constructing a program, and programmers know this. They also know that each one of them has her or his own way of writing software, that discussions about style cannot be settled with technical arguments, and that it might be difficult to convince them about using a given programming style. Paradoxically, they seem to agree that one of the most important things to obtain beautiful programs is consistency ("Consistency and symmetry are the first two design guidelines that come to mind when people think about well-designed software, and of course software has to actually do what it was written to do."[33]) The following quote is an extract of the guiding lines for coding that McCann, a Computer Science teacher at the University of Wisconsin, suggests to his students:

Why Indentation is Important
[...] People have been writing programs for several decades now, so it should be no surprise that some indentation conventions have been adopted over the years. What may be a surprise is that authors of textbooks for beginning programmers rarely devote much time to a discussion of acceptable indentation styles. Instead, they usually adopt a style for their example code and encourage their readers to adopt it as their own.The problem with that approach, as far as I'm concerned, is that it doesn't encourage the beginner to adopt a style that the beginner finds appealing. If you like a style, you are more likely to use it. I'd much rather see you use a consistent, acceptable style that you like and I don't than see you pretend to use a consistent, acceptable style that you hate but I adore. Programming is an art, and its beauty is in the eye of its beholder. Try to find a workable intersection between utility and attractiveness.Having said that, you're much better off finding and adopting an existing style than creating one from scratch. Where can you go to find examples of existing styles?
• Textbooks. Each author has their own style. Try visiting the library and look though some books that cover your programming language. This is particularly helpful when you are using a relatively uncommon construct of the language and you have no idea how to indent it.
• The Internet. Lots of instructors have created lots of web pages for their programming courses, and some of them have taken the time to create documents on programming style, including indentation.
• Also, there do exist some general programming style documents. Visit my Programming Style Documents page for a few links. Even if there isn't one for your particular language, the ideas covered in these documents are usually applicable to other languages as well.
• Language Standards Documents. Your language is almost certainly maintained by an ANSI and/or ISO committee. If you can find a published copy of the language standard document, you can find out what

indentation style they like to use. Often, the style from the standard is the one textbook authors adopt. (McCann 2001)

The issue of consistency can be a nuisance for careless programmers who, even when working alone, change their style within one program, but the real problem seems to arise in projects with several programmers, who might have problems agreeing on a given method ("if you get more than 2 programmers in a room, they'll end up in some stupid religious war over editors or indentation style"[34]). In the following entry *geophile* complains about the difficulty of writing beautiful code and offers some explanations. Explanation #2 is about consistency, is has to do with the 'too-many-cooks' syndrome:

Frustrated artists by **geophile** (#2252963)
In all development shops I've seen it is very difficult to write beautiful code. I find myself getting the job during the day, and writing my own beautiful code, for my own projects, in my spare time. I imagine my feelings about this are not too far from that of a serious musician earning money by playing bar mitzvahs, or novelists writing ad copy. Anyway, here are the reasons why writing beautiful code at work doesn't happen:

• Deadlines. If it ain't broke don't fix it.
• Too many cooks. Someone else touches my code and they don't follow my coding style. Even worse, the offender doesn't follow any consistent style at all.
• The original designer did a crappy job. Which is just another version of If it ain't broke don't fix it.
• No time for re-factoring, even after requirements change, or you just realize better ways to meet the original requirements.

So far I have only discussed about a program's consistency from the perspective of the programming style, but the consistency of a program may also refer to something different, namely to the fact that it behaves in the same way under different conditions. If the conditions vary enormously, for instance a different Operative System, the program will not work at all, but sometimes it is enough to change a few conditions (such as version of the Operative System, Internet connection, etc.) for the results to vary. Programs with this sort of execution problems are not appreciated by programmers: "Visual Basics (particularly VBA) problems include: - Inconsistency (damn thing behaves differently on every computer, application, OS version, etcetera, and the changes in the objects versus the changes in the code are not always coherent)." Naturally, programs that behave well even when conditions are changed are more admired.

ROBUST, STRUCTURED

There are a number of qualities that are also, if less frequently, used to express appreciation. Some of them refer more clearly to a technical feature than, for instance, 'clean', and they are often used as normal (non-aesthetic) descriptors. The statement "this is a structured program" is not necessarily an aesthetic judgement, even if it is a subjective one (there is no objective way of measuring how structured a program is). But subjective and aesthetic are not the same, and we must look at the context to decide whether the adjectives are used aesthetically or not. We saw at the beginning a good example of 'structured' being used in an aesthetic sense, consider this one with 'robust':

If software were a hard science... by **Hard_Code** (#2255196)
...following that analogy, we could build beautiful robust software, in say, a century perhaps? Sorry, but real users want real software yesterday...not beautiful 100% bug free correct software ages from now. Programming != Engineering.

A robust program is one that has been designed to cope with as many unexpected situations as possible. Programs, during their lifetimes, face lots of different situations, many of them practically impossible to foresee (phone lines that are blown away by snowstorms, electrical shutdowns, users that do what they are supposed not to, etc), and only those carefully designed react well (do not fail). Some programs, many programs, are so fragile that they fail without the need of external catastrophes. They fail because the programmer did not do a good job in anticipating the conditions under which they would be used. Or because different programmers worked on different, but interacting, parts without telling each other what they were doing. Or because programmers were not told how they would be used. Or... robustness is a complex issue, and a time-consuming one too.

nice, but welcome back to the real world by **room101** (#2252826)
[...] The bottom line is, software isn't a bridge or a house, people don't trust their lives to my software. If I made software for the medical field or something like that, yes, I would have a different view. But the fact remains that you should only make it bullet-proof when you need to, because you never have time to make everything bullet-proof.

The adjective 'structured' is especially difficult in programming since not only can it be used as a technical descriptor and an appreciative ideal but it also refers to a particular kind of programming, one that can be, a bit simplistically, described as 'not using the instruction GOTO'. There is no need to go into deeper technical details, it is sufficient to know that this kind of structure is an objective characteristic of a program, similar to that of a car engine having V-formed cylinders or not. Naturally, there are no final conclusions as to whether structured programming is more or less beautiful than non-structured programming, very much, I would imagine, like the V-cylinders in an engine. At any rate, in the aesthetic sense that is of interest here, 'structured' means basically 'well designed', or 'well thought-through'.

After going through all these aesthetic ideals it would seem that all software attains one or another kind of beauty, but the reality is rather the opposite. Most of the software seems to be, according to the programmers, rather poor. Whether this is the case or not is of little significance here, the important thing is how they discuss the matter.

They have all kinds of explanations for the perceived abundance of sloppy code, lack of time being perhaps the most frequently mentioned. Indeed, by all accounts there seems to exist an endemic scarcity of time in the software world, being late is what Ellen Ullman calls the "terrible, familiar way of all software" (:3) (Ullman 1997). To those claiming that beautiful software is possible, including Connell, most of the participants answered something like this:

Re: Its a "I'll do it later" thing... by **jmccay** (#2253130)
Either you don't work in the real world, or you have no deadlines. I think a lot of software is "designed bad" for because doing a complete and thorough design which is good, neat, and handles 99% of the problems takes way too much time to fall within the time frame of the usual project. I usually have more than one project going at a time I am lucky if I can afford to work on them for a whole day.Designing "good software" just is not practicle outside the glass house world of education environment. It takes too much time to meet the time requirements of most projects.

This is a typical example of what programmers may say about beautiful software: yeah, it is great but it is unrealistic. Which leaves you with a impression that beautiful software may be more a symbolic entity than something that programmers expect to achieve. Would beautiful software be really achievable if programmers were given more time? Or is it just that having an opinion about software aesthetics (and following one's personal style) is an essential element of 'being a programmer'? It is impossible for me to know whether beautiful software is just some more slack (in the project scheme) away, and it is perhaps also impossible for programmers to know, but what is clear is that the notion of beautiful software is central to the programmers' relationship to software. It serves them, for instance, to position themselves against other groups, such as managers:

Oh Yeah... by **Greyfox** (#2252945)
Try telling your manager sometime that you want to redesign a piece of code because "It's aesthetically displeasing" or because "The design sucks." He'll laugh you out of the office and quite possibly the company. Nevermind that you were right or that your redesign would drastically improve maintability and probably speed things up. Managers don't want good code. They want code that you can squat and shit as quickly as possible because the only metric they look at is the deadline. It may not smell good. It may self destruct in a few months. It will certainly keep your team in "fireman mode" for the rest of the time they're at the company, but by God it made the deadline and that's all that counts.
Just to make matters worse, a lot of managers believe that if they give their programming teams Rational Rose or Visual C++ or whatever, that those tools will magically make the code the team is producing well designed. Well if you give a monkey a computer, he's still a monkey and you won't get anything out of him at the day except a bunch of monkey shit. Most of the commercial code I've ever seen has been monkey shit. Ironically open source code tends to have a much lower monkey shit ratio because the programmers don't have time constrains and care to get their design right.

There will be an opportunity to return to this us-them distinction, since it is such an important part of the private aspects of programming. For the time being we can note that even if lack of time seems to be the reason most frequently employed to explain the whipping up of functions and the production of monkey shit, there are others: some programmers humbly admit their own shortcomings while others complain about a general lack of skills in the field:

Re:Most good programmers are capable... by **mikehunt** (#2255599)
Yeah, but the problem is how few "good" programmers there are. Let's fact it, most programmers write awful code but nobody ever gets to see it. I would estimate that of all the programmers I have worked with over the last 18 years that less than 10% of them could actually write well designed, well structured and maintainable code.

I hope that this long chapter has clarified that 'beautiful software' might mean different things; that, in any case, it is always an expression of appreciation; and also that it can be used to separate programmers from, for instance, managers. All these things form part of the private world of programming, which is the subject of this thesis.

Now, I have so far avoided the most dominant feature of programming, namely the fact that to program is to construct (virtual) machines, hence to construct things that do something, that have a function. It is time to deal with this fact, and to see what it means for the private aspects of programming.

25 It is difficult to exactly describe what “kludge” is, but it is definitely not anything you want to be associated with. Kidders (1981) offers the following image: “*Kludge* is perhaps the most disdainful term in the computer engineer’s vocabulary: it conjures up visions of a machine with wires hanging out of it, of things fastened together with adhesive tape.” (:45)
26 OpenBSD is an volunteer effort (open source) to that produces an operative system (another free alternative like the more known Linux). It has its own website, www.openbsd.org, and in its press coverage section we find the article “Why Linux Will Never Be as Secure as OpenBSD”, whose author, Kurt Seifried, uses ‘clean’ in the following manner: “OpenBSD users on the other hand have an extremely clean and secure code base to work from, that is proactively being audited on a continuous basis.”. Earlier in the article Seifried suggests one of the reasons why Linux is not as clean as OpenBSD: “Linux vendors care about having happy customers. OpenBSD developers don't. The Linux market has become a very competitive space, with around a dozen "major" distributions, and literally dozens (if not hundreds) of smaller players. The major distributions generally pursue similar markets, home desktop users, corporate/educational desktop users and corporate/educational servers. Almost every commercial vendor has invested significant effort in graphical installation programs, desktop software like Gnome and KDE, and other usability/entertainment/productivity software. There is absolutely nothing wrong with this, as more people use Linux the installation must become easier, and things like word processors are needed. However it means that Linux vendors have to spend a lot more effort pleasing users, several distributions now ship on multiple CD's because of all the add on software they include. Although customers complain about security, very few will actually take a secure product instead of an insecure product with more features (even if they may not need those features). Unless a sizable portion of customers start putting their money where their mouth is vendors will not change significantly.” In other words: too many features, too many actors and too many uses, make it difficult to keep the code clean
27 A routine is, for all we need to know, the same thing as a function
28 Commands and operations are not necessarily equivalent. ‘Commands’ refers to the words written by a programmer in the code and ‘operations’ to the orders the processor executes. Only in the case of assembler language are those the same; in other programming languages commands are generally translated (by the compiler or interpreter) into several operations.
29 This is a comment, something the processor will not read. Anonymous Coward is not of the opinion that the lack of comments in Mel’s programs makes them lesser.
30 Slashdot message #483001
31 Slashdot message #2253515
32 Slashdot message #2253618

33 Slashdot message #2255923
34 Slashdot message #2254113

VII
The Relationship between Instrumental and Intrinsic Goodness in Programming

The previous chapter about instrumental, semi-instrumental and intrinsic goodness provided a description of these three concepts with the aim of introducing the notion of instrumental beliefs. This conceptual toolkit was hopefully useful when reading the two empirically intensive chapters that followed. In this chapter we return to the concepts of instrumental and intrinsic goodness (of code), this time using them to explore how programmers approach the relationship between the public and the private aspects of software.

The two most important public characteristics of a program are its function and its price. The function of a program may loosely be described as 'what it makes the processor do'. It is having a function that makes programs valuable and it is this function that characterises them. This is what makes of them tools. And it is, simplifying things a bit, their price that makes them 'products', even if a program has essential economic characteristics other than its price, such as its production and maintenance costs, or its profit.

From the private perspective, programs are, essentially, creations; in the sense that they, above all, reveal things about their creator. Up to this point, I have focused on this essence, setting aside the public aspects of programs. But the fact that code speaks of its creator does not imply that programmers disregard the fact that code is supposed to do something, or that they make a living out of code. In fact, there is a close relationship between the public and the private aspects of code. This relationship will be explored here through a study of how programmers relate the 'intrinsic goodness' and 'instrumental goodness' of the code they write. The study will be based on a classification of programmers according to their approach to the aforementioned relationship. But before the different classes are presented there are a few clarifications to be made.

The first clarification deals with the concepts 'intrinsic' and 'instrumental goodness'. As mentioned above, they are used here as a representatives of the private and public aspects of software, respectively, and their meaning is therefore rather vague. This vagueness is mostly due to the fact that none of those expressions are used by programmers, who instead speak of code as 'beautiful' in the first case and as 'functional', 'useful', or, in some cases, 'that works' in the second one.

‘Instrumental goodness’ is especially vague, since the adjectives it represents (functional, useful and working) actually refer to different characteristics of a program. So different that they can only be grouped together because they are all used in the Slashdot discussions in opposition to beauty (or ‘intrinsic goodness’). In fact, the original title of this chapter was ‘beauty and functionality in programming’. However, this title run into two problems: one, that slashdotters not only used ‘functional’ but also ‘working’, ‘useful’ – and even ‘cheaper’; two, that they did not always use ‘functional’ in a strict sense, but more as an opposition to beauty. They were more interested in explaining the notion of beautiful software than the notion of functional such, which is perfectly sensible, since the discussions dealt with software aesthetics, and not with the strict uses of the adjective ‘functional’. So the title could also have been ‘beauty and its opposites in programming’, but the actual title I think describes better what this chapter is about.

The second clarification has to do with the absence of ‘economic goodness’ from the title. Price being one of the public essences of software, one would also expect it to appear in opposition to beauty. And, in fact, we shall see that some programmers do oppose them (or, if not ‘price’ itself, some other general economic characteristic, such as ‘cost’). I have decided not to include yet another type of goodness because it would make things less readable and because it is unnecessary. From a strict perspective, economic goodness is a kind of instrumental goodness: all we need to do is to think of software as a tool to raise money.

Granted, this is a poor kind of scholasticism, the point is simply that including economic goodness as a separate notion would not have made a better analysis. If ‘functional’, ‘working’ and ‘useful’ can be grouped together under the notion of instrumental goodness, I see little point in leaving ‘inexpensive’ out of it. All four of them are used by programmers to explain their views of beauty.

A third clarification is in order, this one dealing with the correlation between the classifications laid out in the two previous chapters and the one that will be presented here. The message is simple: there are no correlations. There might be some slight correlation between a programmer’s preferred coding style (chapter five) and her aesthetic ideals (chapter six), but nothing worth much attention. At first, my idea was to offer a classification that covered both notions, indeed all thinkable private notions, but it was simply impossible. Constructing such a taxonomy is as hopeless as first classifying painters according to their favourite colour, then according to their favourite subject, and then finally trying to find the correlation between those two.

There are of course other classification principles, and you could argue that I have not found the ones that work. What about, for instance, expert programmers and novices, does this classification have some correlation with aesthetic ideals or coding styles? No, it has not. What about their job positions? or their salary? or their degrees? Nothing. And what about their approach to the relationship between intrinsic and instrumental goodness? Nothing here either. The fact is that there are many different kinds of programmers, with all manners of ideals, styles, approaches, jobs, degrees and salaries, and I have not found any classification that would bring order to these aspects.

Moreover, this is besides the point. The aim of this thesis is not to present classifications of programmers, but to explore a particular phenomenon, or set of phenomena, namely the private aspects of programming. In the course of the exploration I have constructed a few taxonomies, but these are only there to help me explain what these private aspects are, they are means, not ends. It is, as I see it, not problematic at all that the three classifications do not correlate, in fact, it might be an interesting fact in itself: it tells us about the *personal* nature of the relationship between a programmer and her code.

And now, without further discussion, let us examine what different kind of approaches to the relationship between intrinsic and instrumental goodness exist. Please keep in mind that her approach to this relationship will not tell us anything else about a programmer (neither about her aesthetic ideals not about her skills).

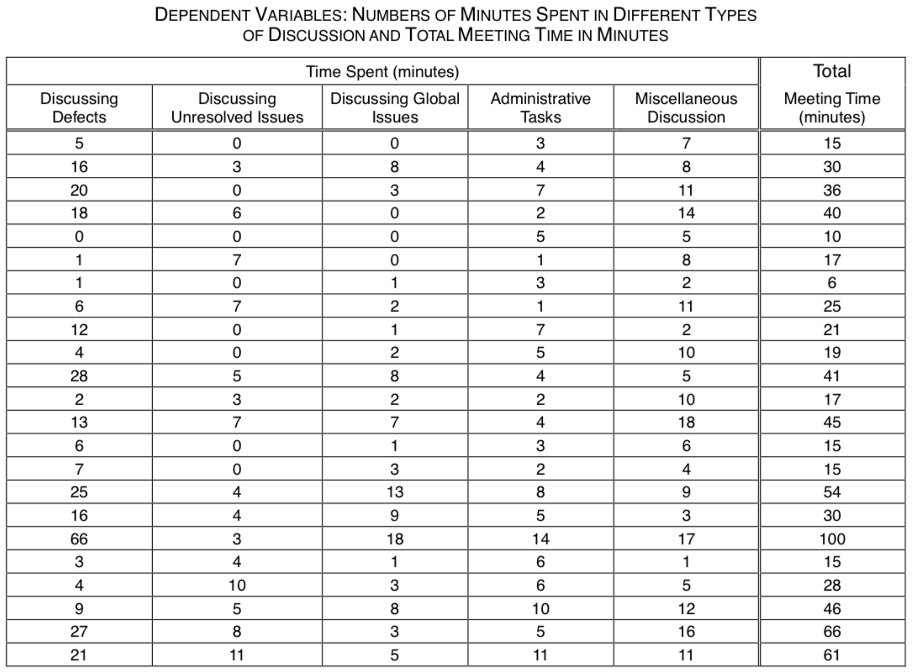
THE ALIEN APPROACH

I start with an ungrounded assumption, since I have not met, or read about, any programmer who would take this approach (hence 'alien'). However, it is an approach that makes sense, so to speak, and it is an approach apparently held by other people. It can be bluntly summarised as 'there is no intrinsic goodness in programming' (alt. there is no such thing as a beautiful program).

This is rather the attitude of those who, in a simplified Heideggerian style (Heidegger and Krell 1993), understand technology as a sterile perception of the world. According to this view, the world, from the technological perspective, appears as 'standing reserve', as a set of possibilities to be manipulated. The essence of the manipulations (technological arrangements) lie in the result obtained, and not in their form; to use the word 'beauty' in this context is the result of a misunderstanding. Hence, when a programmer tries to 'perfect' a piece of code (such as for instance, using well chosen variable names), it can never be a question of enhancing its beauty, only of making it more efficient. The expression 'intrinsic worth of code' is, from this perspective, an oxymoron.

This view seems not uncommon outside the engineering world, and people that dedicate most of their time to perfecting their programming skills and to write exquisite software are not considered artists – as are those who, for instance, dedicate most of their time to practicing Bach's cello suites – but nerds. There have been attempts at introducing the complexities of engineering to the outside world ((Petroski 1992) (Kohanski 2000) (Florman 1994)) – and this thesis is, in some ways, one such attempt – but little has changed. The conventional outside view of software development is blind to its aesthetic dimensions, and main-stream research seems to share this perspective.

As mentioned earlier, the common axiom underlying most of the scientific articles published in specialised journals (IEEE Software, Communications of the ACM, etc.) is that programming can be successfully described as a series of calculations – or objective decisions – devoid of aesthetic value. From this, there is a little step to thinking of it as a process that can be optimised through statistical analysis. Such an assumption results easily in a Tayloristic approach to software development methodologies: the process can be improved with the help of careful time measurements and statistical comparisons. This method does not explain anything about programming, but on the other hand, it assumes that there is nothing to be explained: it is a mechanical procedure, something to measure, not to understand. From this perspective, it makes sense to consider code inspection meetings, for instance, as something whose meaning can be conveyed time measurements (see the figure below).



DEPENDENT VARIABLES: NUMBERS OF MINUTES SPENT IN DIFFERENT TYPES OF DISCUSSION AND TOTAL MEETING TIME IN MINUTES

Time Spent (minutes)					Total
Discussing Defects	Discussing Unresolved Issues	Discussing Global Issues	Administrative Tasks	Miscellaneous Discussion	Meeting Time (minutes)
5	0	0	3	7	15
16	3	8	4	8	30
20	0	3	7	11	36
18	6	0	2	14	40
0	0	0	5	5	10
1	7	0	1	8	17
1	0	1	3	2	6
6	7	2	1	11	25
12	0	1	7	2	21
4	0	2	5	10	19
28	5	8	4	5	41
2	3	2	2	10	17
13	7	7	4	18	45
6	0	1	3	6	15
7	0	3	2	4	15
25	4	13	8	9	54
16	4	9	5	3	30
66	3	18	14	17	100
3	4	1	6	1	15
4	10	3	6	5	28
9	5	8	10	12	46
27	8	3	5	16	66
21	11	5	11	11	61

FIGURE, An alternative description of code inspection meetings as found in (Seaman and Basili 1998)

From this perspective, the idea of making a program beautiful is absurd. I hasten to add, however, that I do not think that researchers such as Seaman do not know that programmers relate to code in terms of beauty, and ugliness. They simply adopt an approach that ignores this fact.

On the other hand, I have met laymen whose concept of programming does not include aesthetic considerations. For them, to insist, as programmers do, that some programs are beautiful and some others ugly is the fruit of a misunderstanding, or of linguistic laziness. If they paid more attention to their language, or to the 'real' meaning of concepts, they would say 'more useful' (or smaller, or more optimised, or faster, or more effective) when they say 'more beautiful'. There are, therefore, no genuine aesthetic ideals, and speaking about beauty in a program is neither legitimate nor illegitimate, it is plain incorrect.

As I mentioned earlier, I have not met, or read about, a programmer who does not acknowledge the possibility of beauty in programming. This does not mean that there are not any programmers who think so, of course. So perhaps the label 'alien' is somewhat extreme… but I still think it is meaningful. More interesting, though, is that acknowledging the possibility of beauty in code – or, more generally, of intrinsic goodness – does not imply that one thinks code should be beautiful. I shall now deal with the different approaches to intrinsic goodness by studying how it is related to instrumental goodness.

INSTRUMENTALISM

This approach is based on a peculiar signifier-signified connection between the two kinds of goodness. Typically, an instrumentalist will say that beauty is a sign of usefulness, or functionality, or 'workingness'. At any rate, a sign of some kind of instrumental quality.

From this perspective, beauty in software does exist, in the sense that some programs arise feelings that can only be described as aesthetic, even if it has little value in itself. The argument is that even if the goal of a program is to work and not to be beautiful, it just so happens that the beautiful programs are the ones that work best. Beauty, or the aesthetic feeling, is a sign of the quality of software; hence, as for instance Connell proposed in his *Software Stinks* (Connell 2001), the quest for beauty is the quest for correctness. Or, as *Ungrounded Lightning* suggests, making it beautiful *is* making it work:

Making it "beautiful" IS making it work by **Ungrounded Lightning** (#2253315)
I don't get paid to create beauty, especially not internal beauty. I need it to work, not look good.
You're paid to make it work, make it keep on working, and do so in an efficient manner.
That is WHY you must "make it beautiful". To do otherwise takes longer and costs more. A LOT more.
One of the problems with this debate is the use of the vocabularity of aesthetics. That software with certain characteristics is also "beautiful" is a side-issue. The characteristics that make it "beautiful" are also those that make it:
fast to write
low in errors
easy to debug
easy to modify, augment, and improve.

Being "beautiful" is pleasant for the programmers (which also improves productivity somewhat). But issues of "beauty" and "style" - AS beauty and style - are a red herring.
And these characteristics that are usually only describable in these terms make an ENORMOUS differece.
For instance: I habitually use these principles in my coding, and debug as I go. My work was once characterized as "... takes three times as long as anyone else, but it usually works."
Bullshit.
The techniques made my work so blazingly fast that I was able to deliver a complete, debugged, essentially error-free component in about three times what it took the other programmers to get to their first clean compile, or to do a debugging iteration. And by "essentially error-free" I mean that in over two years of work at one site, with thousands of lines of code, I had one error detected by someone else, in a preliminary internal release, before I had corrected it.
But the result was that my time-to-completion was measured against everybody else's time-to-do-a-debugging-iteration. And the administration discounted my advice in favor of that of the "most productive" - read least careful - member of the team. And the bulk of the project iterated until the sponsor turned off the money.
So this esperience was an example of how using the vocabulary of art to describe practical issues of programming methodology is actively counter-productive.

Of course we cannot know exactly what happened in that project. Was *Ungrounded Lightning* actually treated unfairly? We do not know, but it is not important. The important thing is how s/he expresses his/her discontent, and the role that the concept of 'beautiful code' plays in his/her misfortunes.

Clearly, *ULightning* does not grant beauty, in itself, much value. In fact, it is "a red herring", and it may be this kind of contempt that feeds the alien attitude. After all, *ULightning's* central message is that programmers that discuss beauty in its own terms are doing everyone else a disservice, they are disconnecting it from its real value: to be a sign of high quality, a sign that the software works properly and is useful. At first sight, s/he seems to be saying that the real meaning of beauty is to be a sign of correctness, a notion that is frontally opposed to the Kantian idea of beauty as an independent quality. Given the fact that the Kantian is, in our society, the 'genuine' meaning, *Ulightning's* apparent suggestion that beauty is a sign of instrumental goodness is a misuse of the word 'beauty'. But those who read *ULightning's* comment as a dismissal of beauty's own value are not reading closely enough.

ULightning seems to be equating beauty with a few apparently prosaic characteristics of code ("fast to write, low in errors, easy to debug, easy to modify, augment, and improve") and one might be tempted to see this as a manifestation of a reduction of the aesthetic feeling. However, if we study those characteristics more carefully, the picture changes. To start off with, all of them share the quality of being unquantifiable and subjective. Yes, one can measure how long it takes to write a program, and how many discovered errors per line it has, so objective comparisons are viable. But visibly, *ULightning* does not suggest a measure of how much time is 'fast' and how many errors are 'low', perhaps because there are no definite measures for this. So “fast to write [and] low in errors” are subjective assessments that require good knowledge both of programming in general and of the specific project in particular. A matter for educated critics, in other words.

Even more difficult to assess are the other characteristics: “easy to debug, [...] modify, augment, and improve.” What is it that makes a program easy to modify? A few things, different for each personality, I imagine, but perhaps all translatable into ‘that the code is readable’... and we have already seen how many different opinions there are on this subject. *ULightning* knows exactly which software is readable for him/her, and hence easy to modify, but this does not mean that everyone else shares that opinion. So those seemingly banal characteristics that made software beautiful turn out to be quite complex; and it would seem that it is a mistake to accuse *ULightning* of dismissing beauty. You can, at most, blame him/her for trying to express it with too blunt technical words. What *ULightning* actually does can in fact be interpreted the other way round: s/he is not reducing beauty to technical characteristics but inferring technical quality from the aesthetic feeling. The process is not “good code implies beauty, and hence beauty is a sign of good code” but “beauty implies (not is a sign of) good code”. It is, therefore, the *look* of the code that decides whether it is good or not, it is from its form that *ULightning* deduces quality. In other words, the instrumental qualities (low in errors, easy to modify, etc.) depend on the intrinsic goodness of code.

An extreme kind of Instrumentalism can be found in the aesthetic movement called ‘Functionalism’. Its goal is to achieve a form that reflects the function of the object; in a way, it is a sort of applied Platonism (cf Plato’s discussion of objects being defective copies of a perfect essence), an aesthetic search of the form that echoes the tool-essence of the object. The aesthetic component of this search is not to find the object's function, which is generally given (a chair's function is to serve as an object to sit on, for instance) to but to find the form that echoes the function. As with *ULightning's* comment, one might try to argue that functionalists reduce beauty to functionality, but what actually happens is that functionalists infer functionality (not function) *from beauty*. The process in this case is not "functional hence beautiful" but "beautiful hence functional."

As far as I have seen, few programmers speak in terms of the code reflecting the function. In fact, most use vague terms. They refer to the general idea of instrumental goodness, using a mixture of concepts, such as workingness (whether a program runs well or not), usefulness and functionality. But sometimes one can find the functionalist ideals very clearly expressed, as when O'Connell wrote, in the article that triggered Slashdot's *Aesthetic Software* discussion (*Software Stinks!*):

The internal design of a software system should reflect and create its external functions. Beautiful buildings merge form and function, and good (beautiful) software does the same. Why does this matter though? As long as a software system works correctly, does it matter what internal design achieves that end? It does matter. An internal structure that acknowledges the external features is more likely to create those features correctly. A software form that follows the software's function also is likely to be simpler, since the external features arise from the internal design rather than being forced on top of the design. A software system whose form does not mirror its function forever will be difficult to debug, will have more bugs, will be difficult to extend and modify, and likely will perform is core functions poorly.

As the attentive reader will have noticed, this approach is, essentially, a manifestation of instrumental beliefs. There are different instrumental beliefs, but they all share the following structure, presented in chapter four:

$$\begin{tikzcd} \text{external quality of program} \\ \text{instrumental belief} \arrow[u] \arrow[d] \\ \text{internal characteristic of code} \end{tikzcd}$$

FIGURE, Structure of instrumental beliefs

This is exactly what Instrumentalism is about: linking an external quality of a program with an internal (formal) characteristic of its code. This process is by no means mechanical and ordered, it happens at various levels, and it has much to do with the vagueness of language. For instance, *ULightning* says that beautiful code is that which is easy to modify. This, obviously, is an opinion, and there is little to say about it. However, one can note that there are different styles of coding and of programming, and that programmers find different kinds of code easier to modify than others. As has been said, *ULightning's* proposition can also be stated 'code that is easy to modify is that which is beautiful'. But is it important that code is easy to modify? Perhaps, or perhaps not. At this level, there is an instrumental belief: *ULightning* believes that easy-to-modify code is better, in an instrumental sense. S/he does not say this explicitly, it is implied in the message, and it is this lack of explicitness that makes instrumental beliefs rather vague affairs. But the results of such beliefs, their manifestations, are quite clear: *ULightning*, for instance, thinks that it is dangerous to speak of beauty in its own terms, and considers it a sign of instrumental qualities. Furthermore, in a discussion such as the one held in Slashdot about *Software*

Aesthetics, s/he will hold that programmers should avoid terms like 'beauty'.

Summarising, the main characteristic of the instrumental approach is that it assumes an immediate link between the intrinsic and instrumental qualities of a program. How this link is actually established (which is the sign and which the signified) is less clear. Instrumentalists suggest that if it is low in errors, easy to modify, etc. then it is beautiful (therefore beauty is a "side-issue"), but I argue that the situation might be the inverse: only after it has been 'decided' that a program is beautiful (any other appreciative term will do), does it become *low* in errors, *easy* to modify etc. In other words: at some point, programmers must decide whether a given piece of code is good enough or not (is low in errors, for instance), or whether a given solution to the problem is better than another; the grounds upon which this decision is made are seldom based on objectively measurable characteristics. They are instead based on their personal beliefs and preferences.

THE 'TRUE-GEEK' APPROACH

This approach, and the next one, are based on the separation of the aesthetic feeling towards the code from the instrumental qualities of the program. Programmers with both attitudes agree that beauty in software does exist, and that it is not directly related to function, correctness, usefulness or price; they differ however in their view on the significance of software aesthetics: is beautiful code something inherently good, what 'real' programming is about; or is it a waste of time and effort?

I shall first turn our my attention to the true-geek approach, which could also have been called "positive Kantian". The adjective 'Kantian' signals the fact that beauty is quite clearly set apart from function – and I say "quite" because they are never perfectly separated in such an instrumental art as programming. The qualifier 'positive' refers to the fact that beauty is spoken of as something good, sometimes as a proof of mastery, but more commonly, as the result of approaching the programming effort with the right frame of mind. The most important difference between this and the previous functional attitude is not the kind of programs that are beautiful – they may very well be the same – but the way the quality of being beautiful is discussed. In this 'true-geek' case, beauty is legitimate, something worth pursuing in itself, for the love of programming, whereas the functional discourse only acknowledges its value as a sign of correctness.

So the functional and this attitude are not opposites of each other, true-geeks do not see a conflict between instrumental qualities and beauty even if they do not equate them either. Instead, they exalt software aesthetics in itself, without making any mention of usefulness, functionality, profitability or any other external characteristic of the code. For instance:

software is like building w/ toothpicks by **MikeFM** (#2254465)
I think in the book 'The Hacker and the Ants' there is a quote along the line of programming being like building out of toothpicks carefully glued together and if just one toothpick is out of place the whole thing comes crumbling down. I always liked that.. it seems very truthful. I might add that programmers are usually encouraged by those they work for to forget careful design and implementation and just duct tape parts together as quickly as they can make it work 'most of the time'. I like to write beautiful code.. as I imagine most real programmers do.. us geeks that live, breath, and dream in code.. but in real life there usually is not enough time or resources given to manage to write really well planned out code. This is why Microsoft sucks and a popular motto is "When it's done!" among the truely geeky programming houses and why open source will eventually kill most commercial software. With commercial software if it's ugly you aren't likely to get a second chance to really make it beautiful. With open source software it may

start out ugly but over time can gradually become beautiful as people clean and fix it. The code is visible and so is everyone elses. You can help each other and learn from each other.

We cannot know, on the basis of this message, what *MikeFM* thinks about the relationship between beauty and usefulness. But it is clear that he thinks it is not absurd to talk about beauty without associating it with functionality, let alone usefulness. "Open Source will eventually kill most commercial software" because it is written more carefully and it "can gradually become more beautiful as people clean and fix it." There is of course an assumption that clean and beautiful software is somehow better (or, at any rate, preferable for the users) than ugly one, but the interesting point is that *MikeFM's* perspective is centred on the programmer and the code, not on the user and the application.

MikeFM's attack on Microsoft – perfectly normal, this being a Slashdot message – allows us to imagine that Windows, for instance, is ugly, but he does not say anything about it being dysfunctional, let alone useless. He simply states that it is ugly, letting everyone understand that ugly software is not what real programming is about. And this is the central element of the true-geek attitude to software aesthetics: beauty and ugliness are not directly related to the program's functionality, or to its usefulness, saleability and profitability, they are instead the fruit of the programmer's approach to the task.

According to true-geeks, one may approach the programming challenge either nonchalantly or with due respect. Sometimes those with the wrong attitude are called 'coders' (or even 'lamers') as opposed to 'programmers'; and, well, there is no doubt about which attitude is the right one:

Code aesthetics by **KingAzzy** (#2252919)
There is a definite difference between a "Programmer" and a "Coder". Programmers are interested in the aesthetics of their engineering as well as the science behind it (the two are non-distinguishable) whereas Coders only care about getting the job done well enough so that they continue to have employment and not get fired.
Programmers are much more expensive than coders and harder, much harder, to find for employment. Coders are very abundant. I have never seen a development department (in the 'big corporate IT world') that had more than just a small handful of true programmers, yet dozens and dozens of coders all whittling away at these massively bloated, poorly designed, inefficient, unscalable, pieces of pure SHIT that absord millions and millions of dollars from the corporate budgets.
I don't think comparing houses and bridges to pieces of software is a very fair comparison, btw.. In construction it's quite easy to put lower skilled people to work effectively for the larger picture (doesn't take much as much skill to lay brick as it does to design the wall) than it does in coding (an inexperienced coder can virtually *infect* the entire project with his or her incompetence.
These are my opinions after working in big IT for too long and perhaps after reading too much Dilbert and Slashdot.

The moral implications of this view of software aesthetics are obvious: the nonchalant approach to programming is bad, it does not acknowledge the exceptional nature of programming, it reduces it to a matter of "just solving the problem":

Its a "I'll do it later" thing... by **FortKnox** (#2252832)
I've found that most of the cause of the problem is people "whip out a function that does that job" so they can compile[35] the program, then never go back and fix it up. Same with code comments. "I'll add good comments later/when I'm done", and you finally get the program stable when it needs to be released.
I find it a ton easier to do everything the way you were taught in software engineering 101. Design the hell outta documents (I, personally, use RUP which I find nice), then code complete objects, nothing that'll just "let me compile", but whole objects. *AND* I'll code in the javadoc when I make the object. The code comes out quit nice that way.

KingAzzy and *FortKnox* are both responding to O'Connell, explaining to him (and the public in general) why there is such a proliferation of ugly software. The latter blames this on an attitude that limits programming to “whip[ing] out a function that does that job”, suggesting that this kind of approach is not going to result in beautiful software. Neither is it the proper thing to do.

The true-geek attitude is the opposite of that: programming should be taken seriously and its significance recognised. The true-geek attitude towards the task at hand, characterised by an acknowledgement of the importance of doing the thing right, regardless of whether it really needs so much attention and care, is not exclusive to programmers. It exists in basically all sorts of instrumental (goal-oriented) activities, since they present an opposition between the goal that must be achieved and the way there: is it enough to reach the goal, or must it also be done with style? Is intrinsic goodness legitimate?

Thorstein Veblen discusses this issue in his work *The Instinct of Workmanship* (Veblen 1990), and his treatment gives the issue an even stronger moralistic position:

Chief among those distinctive dispositions that conduce directly to the material well-being of the race, and therefore to its biological success, is perhaps the instinctive bias here spoken of as the sense of workmanship. The only other instinctive factor of human nature that could with any likelihood dispute this primacy would be the parental bent. (:25)

So, it seems that approaching the task at hand offhandedly is not only going to result in low quality, ugly products that others will have to suffer; it is also unnatural (i.e. against the instincts), and thus doubly questionable in a moral sense. Why, it is comparable to being a bad parent… Our programmers on Slashdot put it more bluntly:

Right now... by **Axe** on Tuesday September 04
..right now I am *supposed* to sit tight and fix a boatload of old, ugly code, apparently written by a crack addict. I know how to make it nice, tight, fast and clean - but they would not let me. Old one passed some joke of a QA, and nobody wants to commit their ass into rewrite - in this times the universal question is "what if it fails and I get laid off".. Sucks. I hate every line I look at, and use every bug as an excuse to clean up part of it..

Axe, apart from struggling with a code s/he finds disgusting, has to fight against those who cannot see the point of rewriting it, as long as it does the job. His/hers message distils the same kind of despise that we saw earlier in *MikeFM's* comment.

MikeFM's disdain for commercial software, or *Fort-Knox´* scorn of the code that just "does that job", can be interpreted as a contemporary echo of Plato's contempt for artists and sophists, two kinds of professionals that, it would seem (Guillet de Monthoux 1998), were more successful than him in moving the masses. It is important to remember that Slashdot is a forum originally populated by Linux developers – it has now widened the audience – for people ready to spend many hours programming for the pleasure of it. Some of them do it also for the cause of a more just world, others because they cannot be kept away from a computer, and others because they'll do anything to combat MS (Moore 2001) (Raymond 1998; Raymond 2000) but most of them seem to share the notion that true programming, and beautiful code, require independence from pure commercial interests; a perspective that has much in common with the conventional, idealised view of 'true-artists'.

Hence, true-geeks (the adjective 'true' starting to sound more and more like a moral qualifier) are likely to experience differences between their idea of proper programming and the world's – particularly the corporate world's – view of the software industry (which can be put into the alien approach class). Differences that have been met by others throughout history and throughout the arts. Guillet de Montoux, who in his *Konstföretaget* (Guillet de Monthoux 1998) tackles the issue of art and business, mentions how Wagner's Bavarian opera-enterprise (operaföretag) infuriates Nietzsche, who originally inspired the whole project into motion but who had nothing but contempt for it once it became a success:

Deceiving transcendentalism... teatrocrathy... something arranged and dishonestly adapted for the masses

The seekers of beauty are likely to be disappointed by those who are ready to make compromises in order to *convince* (another word for 'marketing'). MS might be the paramount example used by true-geeks to illustrate the evil consequences of selling one's soul, or one's programming spirit, but other actors, like managers or even users are also to blame for the ugliness of today's software:

software manager managing bridge architects.. by **Anonymous Coward** (#2252893)
manager -> we need to ship this bridge in 3 months.
engineer -> yes, but it's really big and really important
manager -> yes, but it has to ship in 3 months.
engineer -> so how much weight does it need to support?
manager -> i dunno, I'll let you known in 2.9 months.
engineer -> what is it bridging?
manager -> why all these stupid questions, start building.
engineer -> I should do an architectural drawing first.
manager -> why bother, here's some metal, start slapping it together. Remember it ships in 2 months.
engineer -> I thought you said 3 months?
manager-> oh didn't I tell you, we heard a rumour that a competitor will be shipping their bridge in 2.5 months, so we have to beat them [...]

This anonymous slashdotter is referring to Connell's comparison of software and civil engineering, much to the disadvantage of the former. "We should expect the same level of quality and performance in software we demand in physical construction", Connell said, and this participant answered that the problem did not lie in the programmers, as Connell suggests, but in the managers that dictate the conditions. The main problem with those managers seems to be that they are oblivious to the inner qualities of software, its aesthetic component: for them a program is not valuable in itself, only in relation to its usefulness, preferably measured by its commercial success.

A few remarkable examples of true-geeks can be found in Levy's afore mentioned *Hackers* (Levy 1984): people who would spend entire weeks, sometimes it feels like their entire lives, perfecting impeccably working programs; not because they needed to be improved but just because they *could be* perfected. In the early days of computers at MIT (in the fifties), Levy tells us, there was a particular kind of admired programming stunt: to bum instructions out of a program, i.e. to write the same function with less instructions. At the origin of this behaviour lied the technological fact that computers had very little memory to work with and making your program smaller meant giving more space to others. However, writing short programs became soon more a question of belonging, of being a true-hacker, than of technical considerations.

Various versions of decimal print routines had been around for some months. If you were being deliberately stupid about it, or if you were a genuine moron – an out-and-out "loser" – it might take you a hundred instructions to get the computer to convert machine language to decimal. But any hacker worth his salt could do it in less, and finally, by taking the best of the programs, bumming an instruction here and there, the routine was diminished to about fifty instructions.
After that, things got serious. People would work for hours, seeking a way to do the same thing in fewer lines of code. It became more than a competition, it was a quest. For all the effort expended, no one seemed to be able to ckrack the fifty-line barrier. The question arose whether it was even possible to do it in less. Was there a point beyond which a program could not be bummed?
[…] Jensen […] came up with an algorithm that was able to convert the digits in reverse order but, by some digital sleight of hand, print them out in the proper order. […] *Forty-six instructions*. People would stare at the code and their jaws would drop. Marge Saunders remembers the hackers being unusually quiet for days afterward.
"We knew that was the end of it," Bob Saunders later said. "That was Nirvana." (:45)

It is only natural that programmers who consider that software beauty is worthy in itself, regardless of whether other people are prepared to finance it, or to buy it, become moralists. Theirs is the fight of the faithful, those that see and respect software's 'true' essence, against the sophists, ready to dismiss the most elemental programming etiquette (such as giving it the time it requires) just to sell more. This opposition has, naturally, important consequences in the world of programming, as we shall see in a later section.

THE SOFTWARE-ENGINEERING APPROACH

Re:And How!!! by **Trepidity** (#2253255)
Anyone taken a look at the code of SSLeay? Good package thou).
And that is exactly why I consider software "beauty" to be a minor point of importance. I'd much prefer something like SSLeay to be hideous on the inside but still be a "good package" than some elegant, beautiful, but overall rather useless and crappy piece of software.

Being a software engineer is, in a formal sense, the outcome of a university degree but the software-engineering approach to the relationship between the intrinsic and the instrumental goodness of software is not necessarily related to one's higher education. Programmers do not need to have degrees to hold this attitude nor do all degree-owners share it. I have nevertheless decided to call this the software-engineering attitude because its view of prototypical programming is derived from an idealised image of other engineering disciplines: proper programming must be based on a well defined methodology and on a serious attitude to the task at hand, something that excludes frivolities such as beauty.

This approach is based, therefore, in a separation of instrumental and intrinsic qualities (as in the true-geek case) and its main characteristic is that the search for beauty in software is a waste of resources. Serious programmers should steer clear from any beautifying efforts (any intrinsic considerations) and concentrate on writing software that solves the problem efficiently (i.e. on its instrumental quality).

Re:software is incredibly complex... by **Space_Nerd** (#2253038)
Well, in my experience beauty of code and how efficient that code is do not go hand in hand. The most efficient pieces of code i wrote were butt ugly and needed heavy explanations to my coworkers, but they got the job done in few lines and they took up little resources to do it. So what we want is no beautifull code, but really efficient one, and coding beautifully often goes against it.
On the other hand, beautifull code is easier to maintain and to share, but its always best to have good code, not code that looks good.

Re: nice, but welcome back to the real world by **jbum** (#2252929)
Hear hear. Engineers with an over-developed aesthetic sense are writing their code for other engineers, not the end-user. Too many times in my professional life have I seen inordinate amounts of time wasted on issues which are invisible to the end-user, because some overly-aesthetically minded engineer couldn't sleep at night.It's a craft, not an art; and if you can't sleep at night, try getting laid.

Software-engineers (with a hyphen to avoid confusion with degree holders) are not impressed by software aesthetics, they know that a program can be beautiful, but they consider that programming should not be about creating beauty but about solving people's computing problems. *Laplace's* reaction to Connell's article is a good representative of this:

Ha, but really. . . by **Laplace** (#2252821)
Every extra day that I take to plan, every minute I spend thinking about design, and every extra line of code I write to make my software more pleasing is another line that could add more functionality, another minute wasted not producing something tangible, and another day that I need to be paid. When it comes right down to it, most software is just good enough to get the job done because that is what is most profitable in the short term. I revel in every bit of beautiful code I write, but also know that if I spend too much time making my code beautiful I will be replaced by someone more interested in just getting the job done. If I really wanted to produce art, I would have gone into a field that produces recognizable art.

Both *Laplace* and *jbum* (see above) defend the instrumental essence of software: programs are not supposed to be beautiful, they are supposed to work. *Laplace* even dares, in a true-geek dominated environment like Slashdot, to defend the need for software to be profitable, raising some heat and getting replies of the sort "Are you in management? You sure sound like it." *Laplace's* comment provoked an interesting exchange, but even more interesting is the dispute that ensues in the following comment (the whole thread is available in one of the annexes):

Not this stupid 'programming is art' BS again! by **Flabdabb Hubbard** (#2252879)
to recognize the artistry involved in writing software
What pretentious bullshit. Software is NOT art. It can be closely compared to bricklaying, or cabinet making, it is a CRAFT.
Try expressing an emotion in C++. It cannot be done. Please think before repeating these banal opinions that software is art. It just isn't. Deal with it, and if you want to be an artist, learn to paint.

Flabdabb Hubbard, as *jbum* and *Laplace*, reacts to the idea that software is art, starting off a thread of about 30 messages in which different participants offer opinions on what art is, on the difference between art and craft and on whether programming is any of those. There are all kinds of ideas thrown into the discussion, making classification almost unviable but one thing struck me: many of the comments could be interpreted not as statements of fact but as moral statements. Participants were not discussing what programming is, but what programming should be. From this perspective, *Flabdabb Hubbard's* comment can be read: software should not be treated as art, and programmers should concentrate on writing code that works, not on making it beautiful. Such an opinion is controversial in Slashdot, getting strong support and equally strong criticism:

Re:Not this stupid 'programming is art' BS again! by **chris_mahan** (#2255424)
[...] Computer code is like sheet music. You can have a ten line canon or a 40 pages symphony, either of which looks like complete gibberish to those who can't read sheet music, but which truly represents the art of the artist who wrote it.
[...] Now, I write code. I want to make the user feel a bond with a freaking motherboard. If I succeed in making a grown man or woman "enjoy the interaction" with a piece of plastic/metal/goo, and I have done that on purpose, is that not art?
I contend that in the same way the common man does not recognize Beethoven's 5th symphony by looking at the sheet music, likewise the common man does not recognize great, beautiful, engaging, pleasing software by looking at source code.
There are millions of programmers in the world who consider source code to be art, to be speech. Who are non-programmers to say that it isn't?

The dispute between those who argue that programming is and those who argue it is not art can be read as a debate between the true-geek and the software-engineering attitudes. Software-engineers want to avoid mixing feelings into the work of writing a program since the only expectable result of that is a deviation from the correct attitude, or at least, a loosening of the methodological discipline required for proper programming. They are more likely to put the blame for the overall low quality of today's software on the programmers' shortcomings, rather than on the managers' or users' disinterest, as true-geeks did.

Planning and review save time and money by **tim_maroney** (#2252954)
Every extra day that I take to plan, every minute I spend thinking about design, and every extra line of code I write to make my software more pleasing is another line that could add more functionality, another minute wasted not producing something tangible, and another day that I need to be paid.
That is an absolutely absurd statement. Every moment spent in planning, review, consideration of potential problems, creation of general-purpose solutions, and documentation of architecture pays for itself many times over later in the development, validation, release and maintenance cycles. Failure to undertake sensible planning activities early in a project leads to massive schedule delays from forced late-game

rearchitectures that would have been headed off by early consideration, review and communication.
Software engineering is the only engineering discipline in which the practitioners are permitted to indulge themselves in work without planning or review, and that's the #1 reason that software sucks.
Tim

It may look strange that I put both *tim_maroney* and *Laplace* in the same group, given that the former's comment is an acrid reply to the latter's. But his attack might be the result of a too hasty reading of *Laplace's* comment. It would seem that *Laplace* is defending the lack of planning, but if we read carefully, we see that the only thing s/he says can be dispensed with is the "extra line[s] of code I write to make my software *more pleasing*" (my emphasis). *tim_maroney*, on the other hand, seems to read that *all* planning is unnecessary, and reacts to that. But the part of his message I want to focus on is the last paragraph: "Software engineering is the only engineering discipline in which the practitioners are permitted to indulge themselves in work without planning or review, and that's the #1 reason that software sucks."

tim_maroney exaggerates a bit to make his point (this is perfectly normal in Slashdot – and in life), because the fact is that most programmers do, in the great majority of occasions, plan ahead, at least as much as they are allowed to. More interesting, however, is his idealised view of the methodological approach applied in other engineering disciplines. He is not alone in this, quite a few of the entries to the Slashdot discussion were based on the same idea, some of them suggesting that civil engineering, or vehicle engineering, for instance, are much better planned, among other things, as a result of the liability typical of those industries. This idealisation can surely be explained in part as the fruit of Connell's article, that praised civil engineering as the example to follow ("We should expect the same level of quality and performance in software we demand in physical construction"); after all, the Slashdot discussion I am presenting here was triggered by that article. But, as I see it, the idealisation runs deeper, it has its roots in two main conditions: in the way programming is taught and in the main body of literature on software development projects; both of which are interrelated. In both cases, engineering is idealised as a highly disciplined endeavour, in which strict methodologies should be applied (which may also be partly responsible for the general view that engineering has no private aspects, as we saw in the alien approach).

Against this idealisation of methodology one can present the work of Feyerabend, who denounced the irregular nature of scientific method. If not even an activity as formalised as science proceeds according to a strict methodology, it is reasonable to ask whether engineering actually does, as software-engineers seem to suggest. Indeed, in what I see as a continuation of Feyerabend's arguments, the growing body of Science Studies shows that an essential part of technical and scientific development comprises political struggles, social conventions and taboos, shifting economic expectations, etc ((Latour 1996), (Bijker and Law 1992) are two good examples). These texts and our material indicate that Lindblom's (Lindblom 1959) phenomenon of 'muddling through', originally observed in the administrative environment, also operates in other settings (such as technological creation).

Clearly, thus, the software-engineer approach acknowledges the existence of private aspects of programming, at the same time as it denounces them as a source of poor software. Hence, software-engineers advocate the installation of strict programming methodologies, which should prevent programmers from wasting time in useless aesthetic considerations, and produce better software. However, the link between following a strict methodology and obtaining better software is not so easy to prove. It is based on a belief that relates the instrumental qualities of a program not with the formal characteristics of code (standard instrumental belief) but with the formal characteristics of the programming methodology (there are several methodologies). It is a special kind of instrumental belief.

Software-engineers face a difficult problem since they claim that programming should be about solving people's computing problems, i.e. their goal is to write useful programs. But, as we saw in chapter four, it is difficult for programmers to know whether their programs are useful or not, unless they themselves are the users. Frequently, however, they write programs for others, and in these cases they cannot, strictly speaking, decide whether the programs are useful or not. In order to surpass this problem, they set up a careful methodology, in the belief that by following it they will obtain useful programs. Unfortunately, there is no method that assures the usefulness of a program, due to a number of circumstances that will be discussed in the next chapter.

So, despite their rejection of the intrinsic qualities of code, they also have created one. In their case, the intrinsic value of the code is not based on what the code looks like but on the methodology its creators have followed. Following this methodology is good, in itself (since it is not sure that it will result in useful code). Once again, it may seem a play with words, but it has bearing on the role of the private aspects of programming (or the intrinsic goodness of code) and, hence, in the understanding of the software development process.

Instrumentalists will be guided by their sense of beauty towards good software, even if they avoid the notion of software aesthetics. Software-engineers are outright opposed to the idea of writing beautiful software, but in that process, they create a new kind of intrinsic goodness. True-geeks is the only group that explicitly acknowledges the importance of the private aspects. This, however, should not mislead us into thinking that they are the only ones that care for the code they write. As I have tried to show, intrinsic qualities of code play an important role also when the other two groups develop software.

Now, are there any cases in which the private aspects of programming do not play any role in the development of software? Yes there are. Firstly, there is the possibility of not seeing any sort of intrinsic qualities in your own code. This would imply that you see programming as a job, and your code as something you must produce and for which you are paid. This requires a disinterest for what your code says about you, and, even if I have not read about this kind of programmers, this is perfectly plausible. We discussed this approach in the alien section. Note that this attitude does not imply that you will produce code deemed ugly by others (you do not think in these terms at all).

So, who writes all the ugly code that seems to be out there? O'Connell complained about this, and slashdotters complained with him: most of the code they had run across was definitely ugly. Now, is ugly the same thing for all three groups?

From the instrumental perspective, ugliness is a sign of badness: programs that have many errors, that are difficult to modify, unreadable, etc.. Obviously, true-geeks do not think that code of these characteristics is beautiful, and software-engineers will directly see that the code has been written without following any kind of method. Even if their approaches to the relationship between the intrinsic and the instrumental qualities of software are different, all of them agree that sloppy code is ugly, and that the programs that result are bad.

Sloppy code and bad programs, are produced by programmers with all kinds of approaches to the intrinsic qualities of code. Many of the slashdotters that participated in the *Software Aesthetics* discussion admitted to producing bad software, and blamed all kinds of circumstances. Some blamed the lack of time, others the lack of clear specifications, others shitty management routines, others… others blamed other programmers and their attitude towards programming.

True-geeks may accuse software-engineers of not caring enough about the essence of programming (to write impeccable code), and, vice-versa, serious engineers may accuse the frivolous hackers of not caring enough about the essence of programming (to follow strict methodologies and solve people's problems, not to indulge in one's own desires). In most cases, however, your approach to the intrinsic qualities of software has little, or nothing, to do with the instrumental (and intrinsic) quality of your programs. Much more important is whether you are given the resources needed to produce good software.

So, I do not know which of the three approaches results in better software tools, or in more profitable software products. All groups want to produce good tools and inexpensive products, and I doubt that the classification presented here can help us differentiate the, let us say, successful programmers from the sloppy ones. After all, writing good programs is not only a question of having the 'right' approach to the intrinsic qualities of code but also of skills, time-tables, colleagues, managers, customer's interests, etc. And, of course, there are really poor programmers within all four groups, and also really skilled ones.

Just to hammer the point home that this classification of approaches is not correlated with the classification of aesthetic ideals presented previously, I think it would be sensible to wrap up this chapter with a slight return to one of those ideals. The idea is to show that instrumentalists, true-geeks and software-engineers all might share the ideal of writing, for instance, clean software.

As I said earlier on, I did try to find some correlation, to see if, say, true-geeks did not all share the same ideal (or coding style). But I could not find any such correspondence. They may put some different shade on the meaning of 'tight code', but I came to the conclusion that looking for this kind of connections is quite unfruitful. The main finding, therefore, is that, regardless of their approach to the relationship between intrinsic and instrumental goodness, programmers may admire any aesthetic ideal(s).

What differentiates the approaches is not so much what kind of programs are beautiful, but how this beauty is discussed and legitimised. The vagueness of the terms 'functional', 'careful' and 'proper' – that could hastily represent each one of the attitudes – allows them to include all kinds of aesthetic ideals: simplicity, cleanness, tightness, etc. An instrumentalist may say that a clean program is functional (or works well), a true-geek that it is the result of a respectful approach to the programming task and a software-engineer that it is the result of a well organised project.

So, let us, just for the sake of completeness, look at how 'cleanness' may be interpreted from the three different perspectives (remember that the alien attitude does not think of software in terms of cleanness, or any other aesthetic ideal):

true-geek version of “clean”:

not many by **cabbey** (#483023)
[...] In many of the larger projects though you can ocasionally find bits and pieces of pure poetry in code. There's an example in the Linux kernel, I forget exactly where - maybe in the vmm, where someone took the time to fully digest a rather hairy function and they totally rewrote it without changing the inputs, output, or side-effects in a small clean block of code. These are the folks that turely deserve this shirt.

software-engineer version of “clean”:

Re:software is incredibly complex... by **Fortmain** (#2254219)
Beauty isn't really the issue here, it's maintainability. The current project I'm on, I 'inherited' someone elses butt-ugly code. It did the job fine, but I spent the first six months reading, re-reading, and testing, just so I could understand what the thing was doing. All told, I spent over a year just getting comfortable with the program. Meanwhile, I'm also supposed to be updating this thing for a new release every six months! Every chance I got I did 'code clean-up', fixing things that worked, but were difficult to understand the logic of, or just plain stupid (take a long, often-used routine, and make it in-line everywhere rather than use a function!?).

Re:software is incredibly complex... by **budgenator** (#2254950)
I was trained as a COBOL programmer, it's a language that will not die, mainly because it readable. There is a lot that can be done to other languages to increase readability, but the real need is for clean logic, a clear API, and documentation. When Pro athletes go to training camp every year, they are re-taught the basics. We change the names but they basics remain the same, design, walk-through, code, test, and document.

Re:Beauty for beauty's sake makes crappy software by **Anonymous Coward** (#2253811)
Meeting user requirements often means having a bug-free program which is amenable to future changes (to as great an extent as is practical), in a reasonable amount of time. So, when you write "good, clean code", you in fact do it with the purpose of better satisfying the user, be through a superior product or a quicker development cycle or whatever. So, in fact, you are agreeing with the poster...

instrumental version of “clean”:

Re:And How!!! by **Telek** (#2253566)
Woah tiger... It's very possible to have both, and it's not very difficult to have both either [functionality and beauty]. The best code is the code that you can read and maintain, and is functional. In my experience it's better to write clean code from the beginning, as you'll suffer from fewer bugs and easier maintenance in the future

Clean code = cost savings by **Anonymous Coward** (#2252917)
For all the management out there to keep putting deadlines on things that can't be met. Think about it. If you fix something before it is released, you will save your self thousands techsupport phone calls per release! That saves money!
Clean code means cost savings!

35 *FortKnox* refers to the situation in which a program is more or less finished and a programmer wants to check if it works or not. In order to do that, it must be compiled, which in turns requires all functions to be there, even if they only have been whipped up (or, in some extreme cases, only have been defined, but that is another story)

VIII

Instrumental Beliefs

As advanced in the chapter about instrumental goodness, and as described in those that followed, programmers find themselves making decisions based on a mixture of aesthetic preferences, instrumental beliefs about 'what is best' (for the user, for the company and for their colleagues), and technical knowledge. This may result in behaviour that is difficult to explain unless we accept that programming is not an objective activity but one in which personal factors play an important role. This chapter considers the concept of instrumental beliefs through the study of one such behaviour: careful previous design. Is there a point in calling this rather widespread procedure a ritual?

The concept of instrumental beliefs – that (as we outlined in chapter four) emerges from the analysis of programmers discussions but that programmers definitely do not use – is one of the central components of the private aspects of programming. It links the private and the public spheres, which otherwise are unrelated. For instance, as we saw, they relate formal aspects of code, or of code-writing methods, to the instrumental qualities of programs. In other words, they connect the characteristics of code itself (structure, looks, readability, commenting) with its external qualities (not only usefulness but also price, cost, maintainability).

Instrumental beliefs are not false in the sense that, for instance, readable code does not yield more useful applications. Readable code might very well yield useful applications, but not *necessarily*. Programmers, of course, are aware of this. The problem is that the only aspect of the application they have access to, while writing it, is the code. Most of the time, they cannot possibly know whether, for instance, the application will even reach the user, or if it will be an economic success. They probably do not even know how much it costs to develop the application (their salary may only be a small part of the whole project). It is very likely that no-one in the company knows all of these things, and yet everyone must make decisions. Instrumental beliefs cannot, on the other hand, be scientifically proved (at any rate, I do not see how such a proof could be carried out).

Instrumental beliefs are the result of a simplification of reality: everything else being equal, readable code *is* more useful than non-readable code. In most occasions, this is an imposed simplification: programmers may find themselves with in a situation of severely bounded rationality. Other times, of course, programmers may be quite uninterested about widening those boundaries.

Bounded rationality gives rise to beliefs, forcing actors to make decisions they believe (and hope) are the right ones. Of course, the level of awareness about the assumptions required by the conviction varies from actor to actor, but I think it is safe to claim that programming is not the only environment in which instrumental beliefs play an important role. Many researchers in the field of organisation studies have pointed out the existence of phenomena similar to the instrumental convictions discussed here (even if they do not call them like this), and, generally, of different kinds of private aspects of management. It is important, therefore, not to give the impression that programmers are *particularly* inclined to imagine things, and to guide their professional activities by personal opinions. They are neither better nor worse than the rest of us (Feyerabend has shown how also scientists act upon beliefs).

Even though instrumental beliefs are not exclusive to programmers, it is still justified to study how these are manifested in the world of software. As I was saying, they form an essential part of programming, and more specifically, of its private aspects.

We have already taken up two different beliefs, rather common but by no means held by all programmers: that readable code results in better programs (by way of making them easier to modify) and that code that looks functional results in more useful programs. We shall now study another belief, the special kind mentioned in the previous chapter: the conviction that a particular kind of methodology results in more useful programs. In fact, we shall look at two opposed principles, one states that it is (generally) necessary to carry out a thorough prior design in order to obtain good programs, the other that such a method often results in (next to) useless programs.

The idea is that by comparing these two notions we may gain some further insight into the workings of the private aspects of programming. More specifically, we shall introduce the notion of programming rituals. But let me first offer a quick overview of the contexts of these two beliefs.

THE ISSUE OF PRIOR DESIGN

The original conditions under which programming took shape, in the forties, fifties and even the sixties (Lohr 2002) (Cerruzi 2000) (Levy 1984), marked strongly the way in which it is understood today. Even if they were almost totally different – computers were huge, expensive and unreachable machines, which could not be accessed personally, programmers had to wait for their time-slot (the few machines were generally tightly scheduled) to hand over their programs (punch-hole cards) to the machine operator, who fed them to the computer. Every second of execution was counted, and billed; so there was not much room for fiddling around with the code, things had to work from the outset. Stories abound of hackers who would found ways to reach the computers, explore them, program them, play with them, and who often ran into such high bills that the university had **either to** bar them from the computing centres or to hire them as system administrators. But the normal programmers, with limited access, had to avoid handing in erroneous programs: if anything was wrong with the code nothing could be done on the spot, the whole stack of cards had to be taken back to the desk, revised, and re-punched.

Those programs were generally not too large, and 'careful design' meant basically 'careful programming' and implied making sure the program would run once it was fed to the computer. One did not just try things out to see if they worked, but started with a careful analysis of what the code would do. Such an approach was sensible, given the size of the programs and the economic circumstances. This does not mean that all programs written in that era were bug-free, far from it, but that the absolute priority was to code the program correctly from the very beginning, even before feeding it to the computer, since making modifications was extremely expensive.

Things are different nowadays. To begin with, every programmer works on a computer a million times faster and infinitely more accessible than the first ones. Programming today is also, as it was then, writing commands in order to make the processor carry out calculations, but apart from that, everything else has changed. Programs are generally far larger than the early ones and the programmers' interaction with the computer is wholly different.

As I hope has become clear in the course of this thesis, there is no programming methodology today, nor any software development management methodology, that is the undisputed best one, let alone one that could be properly called scientifically correct. Neither the software industry nor the academics know how to make sure programming projects produce the results expected, in the expected time. This is not for lack of suggestions, in fact, there is a large and growing body of literature dedicated to the issue (Cooper 1999; Gabriel 1996; Hunt and Thomas 2000; McConnell 1996; Winograd 1996; Yourdon 1997) – not to mention all kinds of individual suggestions that abound in the internet –, with varying methodological approaches and proposals. Most of them however, agree on the fundamental problem: much of the software written today is not only of poor technical quality, it does not even serve the user appropriately.

The issue of design is of great importance to all those interested in advancing ways to produce better software. However, the word 'design' is, as one of them puts it, a "big word", meaning that it includes a whole load of different activities. Consequently, we find quite different approaches to the role of design in the production of 'good' software. On the whole, though, one could say that there are two main attitudes to design: one based on a careful and thorough study of the intended functionality of the program, and another one that refuses the possibility, or at least the practicality, of constructing a useful picture of that functionality *before* the program is written and users start to fiddle with it. These are, indeed, our two instrumental beliefs.

The first approach is the offspring of the original notion of programming. It is based on the assumption that it is possible to understand the way in which the program will be used and to prepare a detailed design that takes into account the expected functionality of the program, both from the perspective of technical specifications (what the program must be able to do) and of phenomenological experience (what the user can do *with* the program, and how). Once the programming team, which should include other professionals apart from the programmers, has been able to define both, producing the corresponding documents, the designing can start. And once the design is finished, the coding can follow. And after the coding, or during it, the testing. Once all that is ready the product can be installed (or released): a complete program that due to the detailed prior work – technical specifications and design – will fulfil the user's expectations and become an indispensable tool. Let us call this the traditional approach and the prior work simply 'prior design' (notice that 'prior design' includes more than the plain 'design', used earlier on in this thesis).

The proponents of the second approach (let us call them bricoleurs) do not share the belief in the possibility of producing such a design. They do not deny the possibility of producing a coherent set of technical specifications from which to construct a detailed design. Neither do they think it impossible to produce a design that covers perfectly the technical specifications, and to code from that design. The problem is that technical specifications are seldom fixed once and for all at the beginning of the project. They have a tendency to be modified, either by the final users, who have decided that they wanted something else; by the managers, who are eager to include new features, or to shorten the deadlines; or by the programmers themselves who, as they gain insight in the project, are able to see new ways of solving it. Furthermore, the design is also subject to modifications, often due to similar reasons. But even more important than this is that a program that fulfils the technical specifications may nevertheless not be useful to the users: translating their needs into technical specifications is anything but straightforward[36].

There do not seem to be many cases in which a development project achieves the goal of producing code that perfectly mirrors the technical specifications even if, given a stable and realistic schedule (both quite unlikely, it appears), it should be possible. The real difficulty, as mentioned just above, is to produce a program that actually solves the users' needs.

These problems are not ignored by the traditionalists, everyone seems aware of them; their particularity is that they blame them on originally faulty technical specifications and/or design: an appropriately carried out prior design (including both specifications acquirement and design proper) must take all those circumstances into account. Bricoleurs disagree, they claim the core problem lies in the fact that users cannot possibly describe exactly what they want the program to do, among other things because they do not know what *can* be done, and because once the program is available, their reality changes. Furthermore, users do not always approach the designing effort with a positive attitude: not everyone likes changes, not everyone likes computers and not everyone has time for these things, even if they liked them. The result is a half-hearted attempt to provide thorough specifications, leaving the programmers with just a list of functions instead of a clear and comprehensive documentation about how the application may help final users. “Poor designing strategy and poor designing effort”, the traditionalist could argue; “real life”, the bricoleurs might answer, “this will not change regardless of the intensity of the designing effort: the idea of a careful design is, if not outright unviable, then definitely unpractical.”

Not surprisingly, the bricoleurs call software based on prior design ‘monolithic’ (Gabriel 1996), suggesting with that term the inflexibility that results from an application founded on a set of static technical specifications. Their proposal is to do away with the idea of prior design and instead tackle the project in a gradual manner: talk to the customer, present a coarse design, a working prototype, ask for suggestions, re-design, modify the prototype, test, present, ask again... etc. This way they hope to assure that the final product is useful, if not very carefully designed (the same calculation may be done in different parts of the program, the structure is not perfectly tight, etc.). They are perfectly aware that the result of this kind of work does not comply with software’s traditional values (notice the title of the message):

Worse is Better by **gojomo** (#2254683)
Charles Connell's essay presents appealing ideas; it'd be nice to think software could be "more aesthetic". However, the truth is that Worse Is Better [dreamsongs.com] -- as Richard P. Gabriel will often argue (and other times refute).
Exactly those things which make software "ugly" -- complexity, poor modularity, leaky interfaces, and sacrificing design and comments in the name of rapid deployment -- can also help make it more successful, commercially and socially, in the long run.

Exactly those things which make software "beautiful" -- such as Connell's qualities of "appropriate form", "minimality", "component singularity", "functional locality", "simplicity", and even "readability" -- can in fact make software fail, as "great programmers" spend a bunch of time making "elegant solutions" that never catch fire, because they lack the immediacy and approachability of more haphazard solutions. While this idea may sound like an excuse to avoid doing up-front thought, it's actually a hard-earned lesson that what aesthetically appeals to good, well-intentioned programmers may in fact involve all the "wrong" tradeoffs.
Read all the stuff on this topic at Gabriel's Worse Is Better pages [dreamsongs.com], then revisit Connell's aesthetics peice, and Connell may seem downright naive to you.

Both sides defend their positions with reasonable arguments. Who is right? I don't know. Some programmers seem to be doing fine with the first approach and others with the second. And there is a third group that seems to have trouble regardless of their approach. However, since this thesis is not about suggesting methods to achieve better software but about the private aspects of programming, we shall leave that discussion there and continue with the concept of prior design.

By 'prior design' I do not mean the general and commonsensical idea that it is better to have considered the problem carefully, and produced some sort of plan of action, before the coding starts. 'Prior design' refers instead to the elaboration of a meticulous blueprint based on a set of technical specifications[37] that (a) cannot possibly reflect the users' needs, and (b) cannot be perfectly translated to code. The first circumstance originates in the (almost) insurmountable difference between the world of computational logic and the world of human enterprise. Ellen Ullman, a programmer and writer, relates what it may feel like for a programmer to speak to the final users about the application she is building for them:

I talked, asked questions, but I saw I was operating at a different speed from the people at the table [...] Notch down, I told myself again. *Slow down*. But it was not working. My brain whirred out a stream of logic-speak: "The agency sees the clients records if and only if there is a relationship defined between the agency and the client," I heard myself saying. "By definition, as soon as the client receives services from the agency, the system considers the client to have a relationship with the provider. An internal index is created which represents the relationship." The hospice director closed her eyes to concentrate. She would have smoked if she could have; she looked at me as if through something she had just exhaled." ((Ullman 1997) :13)

The roots of the second problem lie in the vagueness of natural language, and the practical impossibility of translating it to the strict logic of computer languages. Once again Ellen Ullman provides us with an articulate comment on the process of transforming a set of technical specifications (the "system") into code:

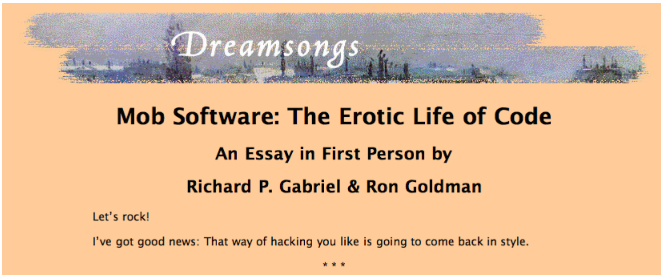
The "system" comes to [the programmers] done on paper, in English. "All" they have to do is write the code. But somewhere in that translation between the paper and the code, the clarity breaks down. [...] As the months of coding go on, the irregularities of human thinking start to emerge. You write some code, and suddendly there are dark, unspecified ideas. All the pages of careful documents, and still, between the sentences, something is missing. Human thinking can skip over a great deal, leap over small misunderstandings, can contain ifs and buts in untroubled corners of the mind. But the machine has no corners. [...] Now starts a process of frustation. The programmer goes back to the analysts with questions, the analysts to the users, the users to their managers, the managers back to the analysts, the analysts to the programmers. It turns out that some things are just not understood. No one knows the answers to some questions. Or worse, there are too many answers. A long list of exceptional situations is revealed, things that occur very rarely but that occur all the same. Should these be programmed? Yes, of course. How else will the system do the work human beings need to accomplish? Details and exceptions accumulate [...] (ibid :21)

So, some programmers (here labelled 'traditionalists') firmly believe that carrying out a prior design results not only in more beautiful but also in better software. Others ('bricoleurs') believe the opposite, namely that prior design usually results in "monolithic" constructions that are difficult to work with and to modify. But these beliefs not only illustrate certainty about different methodologies, they are also, and more importantly, manifestations of a particular view of the world, more specifically a particular view of what programming is, what it should be and what is possible to achieve through it.

This is all very patent in Richard P. Gabriel's arguments against prior design. His thoughts about programming, described in *Patterns of Software* (Gabriel 1996), are inspired in the work of an architect, Christopher Alexander, for whom master plans (more or less the architectural equivalent of prior designs), far from just being neutral construction tools, are the fundamental elements that form a technological structure of control and dominance:

> Master plans have two additional unhealthy characteristics. To begin with, the existence of a master plan alienates the users... After all, the very existence of a master plan means, by definition, that the members of the community can have little impact on the future shape of their community, because most of the important decisions have already been made. In a sense, under a master plan people are living with a frozen future, able to affect onle relatively trivial details. When people lose the sense of responsibility for the environment they live in, and realize that there are merely cogs in someone else's machine, how can they feel any sense of identification with the community, or any sense of purpose there? Second, neither the users nor the key decision makers can visualize the actual implications of the master plan (Alexander 1975, as quoted in (Gabriel 1996))

The traditional view of programming, based on the concept of prior design, has, according to Gabriel, not only alienated users but also impoverished the art of programming. Programming is a human capability that should not be restrained by constricted views about careful planning, Gabriel elaborates his ideas further in an internet essay, co-written with Ron Goldman (Gabriel). The essay starts like this:



Mob Software: The Erotic Life of Code

An Essay in First Person by

Richard P. Gabriel & Ron Goldman

Let's rock!

I've got good news: That way of hacking you like is going to come back in style.

* * *

FIGURE, Programming according to Gabriel & Goldman

The authors, writing in first person, start their essay describing the conditions in which programming is actually carried out. A gloomy view of the present state of software creation:

Over the years I've despaired that the ways we've created to build software matches less and less well the ways that people work effectively. More so, I've grown saddened that we're not building the range of software that we could be, that the full expanse of what computing could do—to enhance human life, to foster our creativity and mental and physical comfort, to liberate us from isolation from knowledge, art, literature, and human contact—is left out of our vision. It seems that high-octane capitalism has acted like an acid or a high heat to curdle and coagulate our ways of building software into islands that limit us.

But like survivors, we've managed to make these islands homes. We've found the succulent but bitter fruit that can sustain us, the small encrusted or overfurred creatures we can eat to survive, the slow-moving and muddy streams from which we drink against the urge to spit it out. Being survivors, we can make do with little. But how little like life is such an existence.

The "small encrusted or overfurred creatures" and the "slow-moving and muddy streams" which programmers must accept are, I assume, the applications from which they make a living. It is the software that stinks, in Connell's terms, forced upon programmers by tough deadlines, clueless management and ignorant colleagues. But most of all, according to Gabriel & Goldman, it is forced upon them by their own fear of failure, manifested in their need for master plans:

Software is full of failure, and will be until we can learn how truly to build it.
Fear of failure is fear of death. In fear of failure, we seek order.
Let me say it plainly: We know how to produce small portions of software using small development teams—up to 10 or so—but we don't know how to make software any larger except by accident or by rough trial and error.—Because the software we're trying to build is too massive—it is simply too difficult to plan it all out, and we have no idea how to coordinate the number of people it takes. Every piece of software built requires tremendous attention to detail and endless fiddling to get right.
In response to this problem we have clung to fads: structured and object-oriented programming, UML, software patterns, and eXtreme Programming[38]. We grasp for mathematics or engineering to come to our rescue—perhaps even the law: By requiring licenses for our developers maybe we can force improvement in software making.
[...] Software development methodologies evolved under this regime along with a mythical belief in master planning. Such beliefs were rooted in an elementary-school-level fiction that great masterpieces were planned, or arose as a by-product of physicists shovelling menial and rote coding tasks to their inferiors in the computing department. Master planning feeds off the desire for order, a desire born of our fear of failure, our fear of death.

Gabriel's suggestion for better software, and more stimulating approaches to programming, are condensed in the concept of 'habitability', which is "the characteristic of source code that enables programmers, coders, bugfixers, and people coming to the code later in its life to understand its constructions and intentions and to change it comfortably and confidently" ((Gabriel

1996):11). But we do ourselves a disservice if we reduce Gabriel's message to its methodological aspects. The opposition between the traditionalists and the bricoloeurs concerns more than just the planification of programming projects. As mentioned above, the opposition goes further, it is two world views that confront each other. Programming is not the same activity, i.e. it does not fill the same place in the great scheme of things, for a traditionalist and for a bricoleur, and the difference between both methodologies can be interpreted as the traces left by this confrontation. We can therefore change the word 'methodology', that only denotes superficial rules of behaviour, with 'ritual', that denotes the processes by which convictions about the great scheme of things are both created and reaffirmed. Or, as Geertz (Geertz 1973) puts it:

> ... for it is in ritual – that is, consecrated behaviour – that this conviction that religious conceptions are veridical and that religious directives are sound is somehow generated. It is some sort of ceremonial form – even if that form be hardly more than the recitation of a myth, the consultation of an oracle, or the decoration of a grave – that the moods and motivations which sacred symbols induce in men and the general conceptions of the order of existence which they formulate for men meet and reinforce one another. In a ritual, the world as lived and the world as imagined, fused under the agency of a single set of symbolic forms, turn out to be the same world, producing thus that idiosyncratic transformation in one's sense of reality to which Santayana[39] refers in my epigraph. (:112)

Geertz is interested in religion, we are interested in programming, and both are obviously not the same thing; among other details, programming does not offer "another world to live in". But they do have some formal similarities, as, for instance, the existence (and the practical effects on everyday activity) of unscientific beliefs. We shall come back to the similarities between programming and religion, but let us leave them aside for the moment and change Geertz' religious conceptions and directives into *programming* conceptions and directives.

Geertz describes rituals as ceremonies in which "the world as lived and the world as imagined, fused under the agency of a single set of symbolic forms, turn out to be the same world" ((Geertz 1973) :112). For those with a traditional approach to programming, prior design is a ritual in which the assumptions about what is possible to do with software (the world as imagined) coincide with what is done (the world as lived). For traditionalists prior design is perfectly reasonable; it is nothing but rational behaviour, considering the structure of the world. The same, of course, goes for the bricoleurs; both groups act rationally, they simply depart from different perceptions of the world.

The symbols used in the programming version of Geertz' fusion (between the world as lived and the world as imagined) are the documents that constitute the design, i.e., the design[40]. This does by no means make prior design an exclusively symbolic activity: its effects have an enormous significance on the instrumental plane. The symbols in this case have as much instrumental as symbolic value; after all, it is during this phase of the software development project that the documents that will be used as the basis for the coding phase are produced: technical specifications, design of user interfaces, database structures, separation between main program and libraries, specification of each library's technical interface, etc. Clearly, all these things are, essentially, instrumental: unlike the case of religious rituals, they are not primarily carried out for their symbolic value. Is it then helpful to say of prior design that it is a ritual? Well, I think so, since it indicates that the production of technical documents (tools) is not its only consequence, even if it may be the dominant (one would have to check if it actually is dominant). At the same time, it is clear that prior design is a very weak kind of ritual: its formal aspects are, in general, not very pronounced and, more importantly, it is not consecrated behaviour

Traditionalists will argue that a correct performance of every phase of prior design results in useful software. For them, this is a true proposition, and bad software can always be explained as the result of some fault in the prior design, some detail will surely always be found... but is it *really* a true proposition? This is a tricky question.

On the one hand, the connection between a proper performance and a desired result are more closely connected in the case of prior design than in the case of, for instance, the ritual of the rain dance; *but usefulness is neither a mechanical nor a logical consequence of prior design* (programmers I have labelled 'bricoleurs' are aware of this, and suggest that the idea of writing 'monolithic' pieces of code be scrapped – their own method does not either logically imply usefulness and is another kind of ritual). On the other hand, analysing prior design from the perspective of ritual may present it in a mystifying light that it does not deserve. Indeed, speaking about *the ritual* of prior design may give the impression that traditionalists are wrong, or more specifically, that they do not ground their decisions on scientifically proved states of fact. It may also create the idea that they write software on whimsical preferences, and that there is (or could be) a way to write software properly, that is, scientifically. This line of thought is flawed on several points. First of all, programming is not a scientific endeavour, it is an act of creation, and it is unclear what it exactly means to carry out an act of creation scientifically. Second, it is impossible, for reasons given before, to identify all the variables that form part of a programming project, let alone to define the relationships between them. Hence, it is absolutely necessary to make assumptions, to have beliefs, about how to write good software. Third (and this is particularly clear from the perspective of rituals), in order to find a different, but still rational, programming methodology we need another view of the world, in which programming is accorded a different nature and a different purpose. Gabriel proposes one such, and his suggested methodology is indeed very unlike the traditionalists'. But are his mob software and his patterns of software just another ritual or are they the real thing?

Clearly, what we need is a world view beyond all the others, from which to judge them. In the West this view is, as has been insinuated, Science. Hence, neither the bricoleurs nor the traditionalists are right until they can prove their propositions scientifically… the problem is that I do not see what this science-of-the-prior-design might look like, even if it appears to me that statistical accounts are not going to take us there.

A discussion about prior design is not complete without at least noting that there is also a Machiavellian way to look at the whole phenomenon. It is not absolutely unlikely that some of those who follow the traditional methodology do so not because they believe it results in better software but because it is the best-established methodology. As such, it is the most legitimate way of developing an application, and if the program proves to be useless, late, unstable or whatever, they can at least guard themselves saying that they are ISO9001 – or something – approved, and that they followed the book. The requirement of holding an instrumental belief is then passed on to the customers, at least if they are to accept that the best way to develop software is to follow the established methodologies. But that is another story.

At any rate, instrumental beliefs are a fundamental part of the private aspects of programming. In previous chapters we touched upon two of them (based on the readability and the functionality of code), without going into much detail. In this one we have studied yet another one (based on the acceptance or rejection of prior design), this time analysing the belief as an element of a particular world view. We have seen how programming methodologies then can be interpreted as rituals in which that world view is reaffirmed.

Programming can, from this perspective, be understood as a symbolic activity, in which programmers express their beliefs about the world, more specifically about the nature and the purpose of programming. The argument continues on this line, and in the next chapter we shall consider a basic form of symbolic activity that can also be identified in programming, and that also has observable effects: sacrifices.

36 Once again, we meet a classification of programmers, this time the two categories are 'traditionalists' and 'bricoleurs'. And once again, the classification is not an end in itself, it is only valid as a means of explaining a particular programming phenomenon (beliefs, in this case). I make no claims as to the comprehensiveness of this classification, there are probably other methodologies, other beliefs about how to write good software. And, once again, this classification is not correlated with the previous ones, in particular not with the one presented in chapter seven. The category of traditionalists is equivalent neither to instrumentalists, nor to software-engineers, or aliens, or true-geeks. All combinations are possible
37 Not all programs have a final *user*, i.e. interact with a human actor. Many of them exist between other programs, where conditions are very stable and where it is possible to write technical specifications that detail exactly what the program must do. In these cases the adjective 'useful' means something completely different to the case in which there are human users involved. The argument here concerns the cases in which there are final, human users
38 eXtreme Programming (XP) is a programming methodology that has become popular lately. It is not 'extreme' in any thrilling sense of the word, as can be seen by reading their principles Jeffries, R. E. 2001 'What is Extreme Programming?'
39 The quote by Santayana to which Geertz refers is: "Any attempt to speak without speaking any particular language is not more hopeless than the attempt to have a religion that shall be no religion in particular.... Thus every living and healthy religion has a marked idiosyncrasy. Its power consists in its special and surprising message and in the bias which that revelation gives to life. The vistas it opens and the mysteries it propounds are another world to live in; and another world to live in – whether we expect ever to pass wholly over into it or no – is what we mean by having a religion." (Santayana quoted in Geertz 1973:112)
40 Unfortunately, both the verb and the substantive are 'design', making it fastidious to express oneself clearly. This is why I have opted for calling the process 'prior design', and the product 'design'. Hence, the ritual is 'prior design' and the symbol, and product, 'design'.

IV
Community

The concept of 'programming community' can be found a little bit everywhere, including scientific articles in the IEEE Software Journal, where it is used to explain why some programmers hold strong opinions on technical subjects that have not been empirically solved. In this chapter we shall explore the relationship between what we have called 'private aspects', our gathering term for that kind of phenomena, and the existence of a programming community. Bataille's concept of sacrifice as expenditure will be our link, the argument being that some of private phenomena of programming (e.g. writing beautiful software) can be interpreted as the manifestation of a constant and individual sacrifice that brings about a sense of community. The sacrifice may take different forms, but in all cases expresses the same: a concern for software itself (its intrinsic qualities), more specifically, an economically oblivious concern for software. Obliviousness, however, is not the same as opposition, sacrifices are not generally carried out in order to waste but in order to express something. In fact, they may make good economic investments, as some programmers insist.

In previous chapters we have observed that programmers face different alternatives when they are programming and that these alternatives cannot be, so to speak, calculated away. They require a personal choice from the programmer. Programmers, we have also seen, are furthermore concerned by what their creations say about them, not only about whether they work or not. This aspect of programming has been here called the aesthetic aspects of programming, which are a part of the private aspects of programming.

Since the same problem can be solved in different ways, the choices made by the programmer, reflected in the code, say something about her. Thus, the code carries a symbolic value that is not a direct function of its instrumental value. In other words, a piece of code says something about its programmer, and what it says about him/her does not exclusively – probably not even mostly – depend on the whether the application is useful or not. The effects of this will be considered in the next chapter.

This chapter is instead dedicated to the concept of programming community, more specifically, to its connection with the private aspects of programming. The argument spins around Bataille's notion of un-economic expenditure (sacrifice) as the origin of the instant of communion. So let us begin with an overview of his concept of sacrifice.

SACRIFICE AS WASTE

The concept of sacrifice and its role in culture has been broadly studied. It is probably the ethnographers that have shown most interest, but I came in contact with it through the work of a sociologist: Georges Bataille. It was while reading the first volume of his classic text *The Accursed Share (TAS)* (Bataille 1988) that I came to think about coding styles, programming aesthetic ideals and other phenomena that will be presented in the next chapter (vanity, disputes about programming languages, etc.) as kinds of sacrifices. The concept of sacrifice can be treated in different ways, and I may as well clarify this from the beginning: my argument is based on Bataille's notion of sacrifice, in fact on a very concrete side of it (it being an un-economic expenditure), and we shall see that my treatment of the concept is not quite as careful as the ethnographers' and anthropologists'. I use it here as a tool to explain an aspect of programming, inspired by Bataille's *visionary* work; the purpose is not to propose an account of sacrifices as detailed as is the norm among ethnographers and anthropologists.

Programmers do not carry out what we conventionally call sacrifices, very much like they did not carry out 'proper' rituals. The other-worldly aspects are missing in both cases. Programming rituals are not sacred in any other sense than the metaphorical, and the sacrifices are not offered to supernatural gods. The idea is that by interpreting some of their actions *as if* they were sacrifices we gain further insight into the meaning of programming. But before going into that, what is a sacrifice, literally? Or, rather, what are its essential characteristics, as they have been discussed by Bataille?

Bataille was particularly interested in the concept of sacrifice, and the first volume of *TAS* is arguably a long description of the notion of sacrifice and its effects on the celebrants. He takes up a few different kinds of sacrifices (Aztec, Tibetan, Christian, etc.) but he is not as interested in enumerating the formal characteristics of the rituals, or in categorising them, as in studying their fundamental essence. His suggestion is that sacrifice must be understood, above all, as the consumption of resources with an uneconomic frame of mind. Hence, in the case of Tibetans, for instance, he does not delay in a description of the formal aspects of religion but analyses the fact that a whole layer of their society – the monks – is sustained *regardless of the economic sense* that their sustenance may make. This is the single most important characteristic of a sacrifice: the denial of economic reason[41]. The whole *TAS* is nothing else than a severe criticism of the Western fixation with the notion of (economic) usefulness, and of the moral dominance that the idea of 'being good for something' (utilitarianism) has reached in our society. This criticism is a recurring theme in *TAS*, but it is perhaps most clearly visible in the chapter where he analyses Christian sacrifices, and their vanishing in our (capitalist) era. For him, the moral predominance of (economic) usefulness results in the marginalisation of sacrifices and, hence, of the single most important feature of a community (Bataille is more poetic, or more baroque, in his formulation: "it is not necessity but its contrary, luxury, that presents living matter and mankind with their fundamental problems"(:12) – luxury being the essence of sacrifice). Why is the offering of sacrifices such an essential part of a community? Because it is in those moments that the community is enacted, that is when the sense of community emerges[42].

The essence of sacrifice is for Bataille something else than following a formal set of procedures, it is instead nothing less than the denial of economic reason. He opposes the notions of productive and unproductive consumption, using words such as expenditure, squandering, dissipation and, above all, waste, to describe the latter kind. The formal aspects of the rituals become therefore a secondary concern, the essence of sacrifice lies in the consumption of resources with a conscious 'dis-expectation' (omission) of economic return.

From this perspective, the contrary of sacrifice is investment. Their opposition does not lie in the fact that an investment always yields a return whereas a sacrifice never results in anything – neither of those propositions are true. It lies instead in the attitude with which they are carried out: an investment is the result of calculations, a sacrifice the result of one's respect to the power of the sacred. To invest is to consume resources with a productive frame of mind, i.e. to introduce them in a system of economic calculations. The actual – as opposed to calculated – process may or may not yield a gain, but the essence of an investment does not lie in a final success but in the nature of the system of which the speculation forms part. To sacrifice is, on the contrary, to consume resources regardless of any calculations: to give them away as a gift.

Scholars have pointed out that sacrifices usually include an expectation of some sort: the Aztecs did not ritually kill children, women and prisoners for pleasure. On the contrary, their world-view (their world as imagined) required them: they were necessary to assure the sympathy of the gods (Carrasco 1999). The difference between this kind of expectation and what I have called economic expectations is the nature of the system of which they form part. An economic gain is expected as the return on an investment, following an economic rationality, based on calculations – which may very well be incorrect. It is impossible to know, in a general sense, which investments will yield a result, and no amount of calculations can change that, but the question here is not whether one can be sure of a later gain but on what fundaments the expectations are based. In the case of an investment, the expectation is based on the hope that the calculations prove correct. The case of sacrifice is diametrically opposed. In this case the expectation is based on the hope of *reciprocation*, the sacrifice is a gift to the gods, who they hope will accept it and give something else in return (like rain, for instance).

As Mauss already made clear in his seminal *The Gift* (Mauss 1967), accepting a gift is part of a social mechanism that includes an obligation to reciprocate it. However, the conditions of this obligation are very vague. There are no laws that govern it other than a general sense of social correctness. Mauss hints at it in his aforementioned work and Lewis Hyde (Hyde 1983) makes the idea explicit: sacrifices are gifts to the gods. These gifts may be made by the celebrants either to reciprocate the gods for everything they have been bestowed upon humans or else in hope that, after accepting it, the gods will feel the obligation to reciprocate. So, in a way, sacrifices are a primitive kind of investment (and given the incertitude involved in the latter, they both share more than is initially apparent), but they differ in an essential point: the expectation of result in sacrifices is not the result of economic calculations, it springs from social considerations.

A gift, including a gift to the gods, is not an economic investment. Certainly one can analyse the economic consequences of offering a gift (what may be expected in return, etc.) but a present given with the intention of a certain return is not a gift, it is an investment (faux-gift?). The boundary is thin, and permeable, especially in a society well trained in reducing things to economic terms, but it exists (Hyde 1983).

A gift is an action governed by social etiquette and not by economic reason (or economic law). Social etiquette consists of rules of behaviour, but both the rules and the punishment for breaking them are vague, an idea already present in Mauss' work. In some occasions, for instance, one is supposed to offer a gift. Going to a birthday party without a present breaks a well-known rule, but it generally does not carry any clear consequences to the offender. One can also accept a present and never return anything, or many years later, or return something of lesser (or higher) value, without for that matter even becoming immoral. There is a social mechanism that rules the giving, acceptance and reciprocation of gifts but there are no automatic levers, no counting, no calculating of returns, no economic considerations.

The kind of ‘sacrifices’ that I have in mind for the programming community are, however, not gifts to the gods, neither in the propitiatory nor in the thanks-giving sense. It seems to me farfetched to set up a series of gods that the programmers, as programmers, relate to, or that form part of a programming world-view. One might wonder if it is correct to name actions ‘sacrifices’ when they have no connection to gods. This depends naturally on the sense of ‘sacrifice’ that one wishes to use and, as I said before, I am following quite closely Bataille’s notion of sacrifice as uneconomical consumption, which, strictly speaking, requires no gods. The sacrifice, in this sense, is an action that occurs among humans, and that has concrete consequences on their interaction, more specifically, on the meaning they attribute to their interaction.

The suggested programming ‘sacrifices’ differ from the conventional ones in that their formal structure is little or not developed at all. Hurbert and Mauss’ (Hubert and Mauss 1964) offer a structuralist approach to sacrifices, suggesting they can be divided into various moments. I make no claims as to the existence of these moments in what I call the programming sacrifices. This is a metaphor, and I am only interested in illuminating one aspect of the programming effort. More specifically, I am interested in those moments of waste (uneconomic consumption of resources), which, according to Bataille, are the moments of actual *community.* The actions that temporarily transform a group of programmers (which initially are an instrumental system, for instance a programming team) into something that transcends the mere economical convenience are what I have called 'programming sacrifices'. The denial of economic reason that lies at the foundation of the sacrifice gives rise to the moment of communion.

Denial of economic reason does, however, not imply denial of practicality. Due to its visionary nature, Bataille's work lends itself to different interpretations, and to critiques, such as Daniel Miller's: "Bataille was wrong, above all, because his vision of sacrifice was one of pointlessness that thereby repudiates utility, but in traditional sacrifice … the dominant concern is to achieve specific purposes – which are often pragmatic and practical." (Miller 1998) As I see it, though, Bataille does not deny the idea that sacrifices are pragmatic and practical but the kind of pragmatism they are based on. Bataille simply highlights the non-economic nature of the pragmatism that lies behind sacrifices. Pragmatism and economic reason are not synonymous, and this is the main point of the programming sacrifices.

BEAUTIFUL CODE AS A SACRIFICE

In what sense do I then mean that programmers carry out sacrifices? There are no gods, no reciprocation and no propitiation, no formalised destruction of resources and no public performances. But I suggest, with Bataille, that there is something even more essential to a sacrifice than all that: the dissipation of resources, and the sense of community that this waste brings about.

Take, for instance, the notion of beautiful code. Programmers do not perform rituals of computer burning, or anything as extravagant as that. Their offering is in accordance with the immaterial nature of the fruits of their work: the main resource they waste is time (it is difficult to see how knowledge could be wasted… other than by not being applied, but this is not what I have in mind); the time dedicated to keeping a consistent programming style, to making their programs tighter, more habitable, more elegant. Also the time dedicated to "parental-visit-strength clean-up", to proper indentation, to the selection of fitting names for their classes, functions and variables, to the formulation of their comments, to the search for the right command, or function, etc.

Suggesting that we consider these activities as sacrifice is not at all unproblematic. As I said earlier, they do not follow the phases proper to a sacrifice, as described by Hurbert and Mauss. There is, for instance, no phase one, in which the programmer turns her/his attention to the offering that is going to be made, to the gods, to the sacredness of the moment. All those activities (naming, indenting, finding existing commands and functions, etc.) are part of the programming process and are often undistinguishable from the non-sacrificial components of programming: constructing an algorithm, designing the structure of a database, etc. In fact, it is not that they are undistinguishable, it is that they are two aspects of the same activity. Constructing an algorithm cannot be divided into 'constructing the algorithm proper' and 'making sure it is elegant'. Both activities are the same. Either the algorithm is elegant or it isn't. To perfect one that already exists is, in fact, to construct a new one.

Remember that we have considered what programmers say about programming, I have not conducted an ethnographical study in situ. But even if I had, I do not think (based on my short experience as one) that it is possible to differentiate specific sacrificial frames of minds. A programmer sitting in front of a computer may stop typing for a second, try to remember that little neat function, what its grammar is, in which library it can be found, in what cases it can be used... Perhaps s/he will have to ask a colleague, look it up in a manual or post a question on the internet; time will be spent on the issue but it is incorrect to think of it as 'writing the program' + 'making it more beautiful by adding this neat function'. Writing the program *is* adding this neat function, there is no separation. It is hard to see, just by looking at the process of writing the program, that anything worth the name 'sacrifice' is actually going on.

But something *is* happening as the programmers write on, even if it takes something out of the ordinary to bring it up to the surface. One of those extraordinary things is the challenge of going through someone else's code. Programmers become then aware of the stylistic choices of the original author in the code they are handed. They can see whether these little neat functions have been used, whether care has been taken to comment properly, the idea behind the design, the indentation and naming schemes… the readers can also see whether the original author's ideas of 'properly commenting' and 'neat functions' resonate with theirs. Or if they are ugly, or simply sloppy.

The sacrifice in programming is not a separate act that brings about a spiritual transition marked by a series of moments: from A to E, peaking in the act of destruction of resources. The 'sacrifices' carried out by programmers consist of a continuous, more or less conscious, attitude towards the intrinsic qualities of the program they are writing. Maintaining this attitude, and hence producing intrinsically good (beautiful, for instance) software, requires the dissipation of resources (time) and it leaves traces that can be read in the code: from the superficial marks of coding style (the choice of variable names, indentation, comments) to the marks that tell about the program's structure (the abstraction depth of the procedures and the functions, the way in which the main program interacts with the rest of the elements, the programming languages, etc).

But is there any waste at all in the maintenance of this programming attitude? The fundamental characteristic of sacrifice, as it is used in this thesis, is the dissipation of resources. If writing beautiful software cannot be separated from writing software then where is the denial of economic reason? Well, listen to the programmers' complaints about being forced to write software that sells, as opposed to beautiful software:

It's the buyers fault, not the programmers fault. by **maitas** (#2254987)
Crappy programmers are cheaper than good ones. People prefers to buy cheap software with lots of features, even if it doesn't work!! So right now the situation is that, or either you make crappy cheap software with lots of features fast and keep selling like crazy (Microsoft way), or you built expensive great code with only a fraction of the features (belive me, to add features to a given soft takes lots of time) and no single person will buy it.
!!Don't kill the messanger!! Kill the software buyers!!

Re:complexity by **StevenMaurer** (#2253780)
That's an easy one to answer. Software companies don't back their products because their customers don't expect them to. Instead they expect low prices.
Bill Gates is a billionare because he was one of the first people to realize that given a choice between a $200 program that works flawlessly and a $99 program that fails 5% of the time, most people (and businesses) will choose the cheaper product (while moaning how *bad* software is). [...]

features vs bugs by **josepha48** (#2252949)
The problem is that while software developers may want to fix the bugs and make it work nice and all, the managers generally want to make money and the only way to sell a product is through new features. Usually adding in features after an application has been developed makes an app a nightmare to work on and harder to debug. [...]

Not a hill of beans. by **Anonymous Coward** (#2253224)
I look forwards to the general public gaining an appreciation of good code. Much of the discussion so far seems to revolve around how software is not hardware and how much harder it is than physical engineering. No comment on this issue. [...]
The point for me is that the general programming community (I) need to successfully explain to the general public (my mum) why some code is beautiful and other code is not. Until then the public will not pay more for good code than bad code. I am failing miserably so far. [...]

The constraints that economic (costs, profit, competition) considerations put on programming are experienced by most programmers as obstacles to beautiful software. There is however no necessary connection here: not all profitable software is ugly and not all ugly software is profitable, regardless of the quite radical opinions held by some programmers about this matter (particularly about Microsoft’s products). My point is not that commercial software (as opposed to open source, developed without economic constraints) is necessarily ugly, or that open source programming will necessarily yield better software, but that (some) programmers experience an opposition between economic constraints and beautiful software.

However, this opposition is not sustained in the same terms by everyone. In some cases it is simply rejected, as *Myopic* does here:

Re:nice, but welcome back to the real world by **Myopic** (#2253074)
I've seen this comment a lot in this discussion: "well, my software works, so it's good enough". You even say that you don't get paid to make pretty software; just usable software.
I suppose that might be true, but I would venture that not everyone is in the same boat. I, for example, AM paid to write pretty code. My job is to come up with relatively simple perl scripts (modules) to solve various problems that Dartmouth [dartmouth.edu]'s website users have. (For example, I wrote a quota module to help people verify that files they want to write to disk will fit within their alloted disk quota.)
I have NEVER turned in to my boss anything but well-documented, well-commented, readable code. I don't do this out of respect for my users; frankly, I know how to use the software and if they don't they can read my docs and try to figure it out. No, I do it for the other schmucks like me. At some point, my boss will probably tell his next lackey to add some little feature to one of my modules, as he's asked me to do with some older programmer's works. And it's DAMNED IMPOSSIBLE to wrap my head around code which is all mixed up. I comment for other programmers. People who might need to sink their hands into my code.
Paying me now to write comments and format things well is worth it for the added speed with which the software will be maintained in the future. So for me, and I'm sure most of the code jockeys on Slashdot,

the "real world" is one where software is written, THEN MAINTAINED. Beauty is part of maintanenence.

As *Myopic* argues at the end of the message, some programmers seem indeed to be of the opinion that truly beautiful software is more profitable, defending their ideas with cost analysis:

Re:software is incredibly complex... by **Anonymous Coward** (#2253398)
That's about the most naive blanket statement I've ever read. "its always best to have good code, not code that looks good."
Consider a project that lasts 5+ years. Over the life of the project, there will be dozens of developers added and cut from the payroll. Assume your "good code" gets executed once or twice a week and instead of taking 2sec it takes 1sec. You've saved 1sec (possibly 2sec) per week which adds up to 52secs (or 104secs) per year.
Let's compare that to the human maintainer. Assume one person has to look at that code 1 time every 6 months and it takes them 30 minutes to understand it. That's 60 minutes that someone is getting paid to understand that code.
If the person earns $80,000 US, that's about $4 (assuming 4 week vacation) that was spent on the human. It's actually less when you consider that a person's salary is not their labor rate. Over the life of the project (5 years) you've spent $83 on your "enhancement".
Now, if the cost of up'ing the Mhz on your CPU is greater than $83 then it may make sense to implement the "enhancement". However, when you consider the price differential between a 900mhz processor vs a 933mhz processor (the argument being that a 933mhz processor could run the slow code and keep up with the 900mhz processor running the fast code) you won't find an $83 difference. It'll be more like $20.
That being the case, humans are more expensive than computer CPUs these days. Maybe they weren't in the past, but they are today.
Another argument for clean easily-readable and understandable code is that, if you take your argument, the entire system will become "enhanced" and no one will understand how it works. That will add on an additional overhead in the form of lack-of-enthusiasm for a project and will have financial implications.
All in all, I've worked on XP projects where code formatting and understanding was important. And I've worked on government contracts where people hack'ed their way through to save a couple of cycles. Maintainability speaks volumes... And I'd go with readability and understandability in a heartbeat...

The idea is that managers who push for tough deadlines in order to be the first to market make an economic miscalculation: the software will become more expensive as bugs have to be corrected, angry customers met, etc.

Re:No financial incentive for good software by **mrbuckles** (#2253107)
Actually, I would argue that there is a very real financial incentive to well designed software. Namely, it is easier to maintain and extend. The problem I've seen (and this is from building in-house software for, say, banks) is that the managers of software projects don't understand this fact. They only understand the known relationship between time spent on a project and cost. They don't figure the time that will be spent AFTER initial coding.
The amazing thing about all of that is that there are thousands and thousands of pages of studies done on these very topics. Books are written every year to discuss this. It's a field of study unto itself. Yet, most managers you work for will still believe if you're behind on a project you should double the number of people working on it to get done twice as fast. At some point, the responsibility for these problems needs to be pushed up to people who can do something about it.

Clean code = cost savings by **Anonymous Coward** (#2252917)
For all the management out there to keep putting deadlines on things that can't be met. Think about it. If you fix something before it is released, you will save your self thousands techsupport phone calls per release! That saves money!
Clean code means cost savings

Would commercial software houses make more money if they wrote more beautiful code? That is difficult to know, not only because it is hard to know which version of 'beautiful code' to choose but also because market processes are generally too complex to allow for the claim that elegant software will yield higher gains. The anonymous participant and *mrbuckles* are, it appears to me, rather expressing their frustration over the poor quality of software produced and the amounts of time spent correcting mistakes that could – they firmly believe – have been avoided in the first place. Whether or not that would imply any actual profit increase depends on a good deal many more variables.

It is clear that programming sacrifices are not absolute phenomena, and that this is simply an analytical concept. Some programmers can be said to carry out sacrifices, others to carry out semi-sacrifices and others not to carry them at all. The key to the sacrifice lies in an uneconomic concern for the intrinsic qualities of software, and we have seen that the uneconomic nature of the concern may be concealed, or polluted, by a legitimising discourse that follows economic reason. Or else that nature is neither concealed nor polluted but simply non-existent: it is possible that some programmers' concern for the intrinsic characteristics of code is the result of an investment-like calculation. They want their code beautiful because they have calculated that it will yield better programs.

Programmers of this kind are probably uncommon, since those in charge of the economic aspects of software (planning, development, distribution, marketing, sales, services, etc.) do not seem to accept the result of those calculations. It does not sound too controversial, hence, to suggest that those interested in developing beautiful software are not necessarily in phase (do not share the same world-view) with those interested in making economically based decisions about software development. This idea can also be expressed in other terms: considering code something worth in itself is not compatible with considering it just a cog in the machinery of profit.

At any rate, the concept of programming sacrifices is not meant as a categorical truth, it is only a tool for gaining insights into what programming is. There are certain phenomena that can be explained by using such a concept, even if there are certainly other ways to explain them. One of the phenomena that the concept of sacrifice explains is that of the programming community.

One of the problems we meet when trying to apply the concept of 'sacrifices' to programming is the absence of sacred actors: to whom do programmers offer their sacrifices? But even if there are no properly religious parts to be played in the programming sacrifices, there still exists a transcendental actor. An actor that, like the gods in the conventional sacrifice, is *constituted* in the act of writing elegant code. This actor is the programming community. According to Bataille, it is the act of unproductive expenditure, the sacrifice, that brings about the moment of communion, the community. This moment of communion is what puts the participants in the sacrifice in contact with the transcendent realm, and in true contact among themselves.

His idea of community should not be confused with the more usual sense that the word has. For him, community is not about a group of people that have something in common, that meet every now and then, or that live in the same area. That is a community, something that can be founded, whose regulations can be written down, and that exists regardless of any unproductive expenditures. Like Bataille's community, it must be maintained (although not through the offering of sacrifices) but the essence of this 'conventional' community is a certain practical convenience: it is set up in order to meet concrete, including economic, goals.

Bataille's community, on the other hand, is a special frame of mind, or frame of spirit, that is triggered by a common sacrifice. But both kinds are not incompatible, the only thing that Bataille wants to point out with his 'true' community is the existence of a common sense of belonging that goes beyond mere economic convenience. Most 'true' communities also present the 'usual' features: a hierarchical structure, a set of routines, etc. but Bataille insists that the heart of a community lies in the sacrifices, and not in its organisational details; as we saw earlier on, that "it is not necessity but its contrary, luxury, that presents living matter and mankind with their fundamental problems."

The more prominent examples of communion moments are those – archaic – massively attended sacrifices, such as the midsummer's and midwinter's celebrations. Or the Aztec sacrifices, or the native American Potlatch. In all of them, goods gathered by the community are burnt, killed, thrown to the sea or dissipated in other striking ways, which is what makes them so remarkable. But there are other kinds of sacrifices, less conspicuous, which can nevertheless be described as unproductive expenditure. Small everyday offers, like fruits, tea leaves, etc (Plog and Bates 1976), that also put the person that carries out the offer in contact with the transcendental realm.

The programming 'sacrifices' follow the second form. The unproductive consumption of resources that takes place while programming consists of inconspicuous actions – perfectly camouflaged in the normal flow of events – that keep alive the sense of community among programmers.

The programming community emerges as a by-product of each member's concern for her code's intrinsic value, and of the actions that ensue from such a concern. The efforts of making it beautiful, what I have called 'programming sacrifices' create a sense of communion, regardless of their economic effects (profit, user-happiness, etc) – or I should say, exactly because of this "regardless". It does not matter that there are different opinions as to what makes software beautiful, the fundamental issue is that it is worth making it beautiful.

This moment of communion is made durable, and observable, through, for instance, the (in)formal communities that emerge around topics of interest to programmers. Slashdot, the website from which so much of my empirical material comes, is a good example of such a community (the expression 'Slashdot community' appears every now and then). In fact, websites are a frequent form of manifestation, and there are sites dedicated to Perl, C, Assembler, eXtreme Programming, Unix, Linux… anything. The 'sacrifices' (uneconomic waste) carried out at these places include activities other than writing elegant software such as offering free technical advice and free code[43].

The more general, and more loose community of programmers (as opposed to the more concrete examples given above) does not present manifestations as obvious as websites. There is no www.programmer.com, perhaps because programmers speak all kinds of languages. Perhaps also because the 'sacrifices' they all perform, their communion link, is not experienced as such: the idea that code is worth in itself is not strong enough to generate organisations and routines (to make itself durable).

But there is a common, and vague sense of forming part of something which has been noted for instance by Sharp et al. while studying programmers' conventions. In their article *Software Engineering: community and culture* (Sharp, et al. 2000), they describe 'un-scientific' behaviour among programmers, who may, for instance, consider particular languages "undesirable without recourse to any justification or explanation", and which may be ridiculed in public at a conference "devoted to choosing among languages for teaching":

> We see a surprising amount of movement towards C++. Devastating really because if you ask them why, they haven't got a clue (audience laughter), not a clue. We see all sorts of anachronisms; people still programming away in Basic, Fortran, or SQL. Why (audience laughter), for goodness' sakes, why?

Clearly, the speaker (and it would seem part of the audience too) has a definite opinion about someone who chooses Basic to teach programming. Not particularly positive, it appears. What this opinion is based on is hard to know, but it probably has more to do with personal experiences and preferences (and perhaps with a desire not to appear old-fashioned) than with a scientific proof that Basic is, indeed, a bad alternative when teaching programming. Sharp et al. ascribe this behaviour to the existence of a "distinct culture of software engineering [that] transcends national, regional and organizational cultures" (:40). I would rather go the other way round, unscientific (and uneconomic) ways like the one described above is what constitute that "distinct culture" (sense of community).

As an end to this chapter, I would like to present one more manifestation of the existence of a programming community. It is based on material from the Slashdot discussions, but I think it is not too farfetched to imagine that similar manifestations take place among other programmers. Let us now see how programmers moralise about those who do not show concern for software's intrinsic qualities (or, in other words, about those who are not ready to carry out programming sacrifices).

MORALISING

The programming community, as I use the term, is a weak phenomenon; it is as weak as the sacrifices are unplanned and lacking in common rituals. Those programmers that can be said to carry out sacrifices do absolutely not think in terms of sacrifice, they think in terms of elegant problem solving. They approach their computing problems with a vague vision of what kind of programs they seek to achieve (clean, tight, scalable and other ideals). Despite the lack of public sacrifices, the communion does exist and we can observe some of its effects. One of the most visible traces it leaves is the moralising that takes place among programmers. This process, which deserves a more thorough study than the one I propose here, classifies people into the good ones, the bad ones and the clueless ones, who are mislead by the bad ones to side with them and who, inadvertently, contribute to the expansion of evil software (programmers' moralising schemes are as simplifying as anyone else's). The good ones are the programmers who fight, against all odds, for the creation of good software. As we have seen, there are different opinions as to what makes software good (see chapter seven), but all share the view that writing good software is the duty of every 'true' programmer.

True-geeks, instrumentalists and software-engineers alike moralise about how programming should be done, and the following quotes come from all three groups. For instance, do you remember *MikeFM*:

software is like building w/ toothpicks by **MikeFM** (#2254465)
I think in the book 'The Hacker and the Ants' there is a quote along the line of programming being like building out of toothpicks carefully glued together and if just one toothpick is out of place the whole thing comes crumbling down. I always liked that.. it seems very truthful. I might add that programmers are usually encouraged by those they work for to forget careful design and implementation and just duct tape parts together as quickly as they can make it work 'most of the time'. I like to write beautiful code.. as I imagine most real programmers do.. us geeks that live, breath, and dream in code.. but in real life there usually is not enough time or resources given to manage to write really well planned out code. This is why Microsoft sucks and a popular motto is "When it's done!" among the truely geeky programming houses and why open source will eventually kill most commercial software.
With commercial software if it's ugly you aren't likely to get a second chance to really make it beautiful. With open source software it may start out ugly but over time can gradually become beautiful as people clean and fix it. The code is visible and so is everyone elses. You can help each other and learn from each other.

As we saw, true-geeks accuse software-engineers of not doing real programming, and vice-versa. But apart from that, they have common foes. Among these we may emphasize the actors of the software industry that are only interested in the profit-aspect of programs: reducing software to profits denies that code is something worth in itself.

Even true programmers understand that software must be sold, but they argue that programming should not be forced to serve economic objectives, high quality should be given priority before marketing goals such as time-to-market and price. The clash between the intrinsic and instrumental (including economic) perspectives on software gives rise to many comments:

Re:software is incredibly complex... by **sg_oneill** (#2255418)
Yeah bloke, I sorta agree, but the thing I note is your refering to 5 year plan projects, and not everyone is talking in that sort of headspace where we can do the whole waterfull-pretty diagram-dfd-usercase-point'o'failure analysis mumbo.
In the industry I'm in, I'm more likely to be hit by management with the "How long will this take?" ME(After back of envelope figurin' "Month & Half to do it properly" Manager: "You've got a week". A lot of programmers get that sorta thing. Granted that there is gonna be a little noodling with a rough sketch of how it's gonna hang to gether, quite often it's a rough job on an ill-considered designed followed by fixum-hacks because commisioning is tomorrow.
And yeah... The bugs then roll in. I've figured that for every day stripped of a sugested timetable for development, three days are added fixing the mess.
But don't blame the programmers. The "Fixit or fired" managerial aproach kinda forces it
Reverse engineering documents can wait. Commenting'll probably never happen.
It's a shame, but that's life in small business.

So the ‘bad’ people are those who refuse to see the intrinsic qualities of software, those who measure everything in money. ‘Managers’ are an easy target, they are the ones who most obviously do not care about the code itself, only about its sales. But they are not the only culprits, users are also to blame for neglecting quality and focusing on prices:

BiggerBetterFasterNow! by **lupine** (#2253967)
[...] The general lack of quality in current software apps is a reflection of our society. Quality Craftsmanship is a thing of the past, mass production is where its at. Businesses and managers are pressured to bring products and services to market As Soon As Possible. Process design, peer review, and quality control of central components fall by the wayside as new useless features are added willy nilly in an effort to bolster sales. I try to write good software, but without proper project planning there is never enough time or thought allowed to enter the process. Sometimes I feel like writing quality commercial software is like trying to swim upstream.
I have written some good tools that are used by many of our applications to quickly build linked customizable html displays of database data. I used another project as an excuse to build the tools. No time in the workplan for that! And have never been allowed time update and extend the tools even though it would clearly be beneficial and save time for many other programmers. Projects could be created with object oriented reusable code and modules well planed, organized, optimized etc, but clean code doesn't sell.

Consumer driven, not quality driven. by **tshak** (#2254328)
We should expect the same level of quality and performance in software we demand in physical construction. Consumers are not willing to pay for such quality, or wait for it.

It's buyers fault, not programmers fault. by **maitas** (#2254998)
Crappy programmers are cheaper than good ones. People prefers to buy cheap software with lots of features, even if it doesn't work!! So right now the situation is that, or either you make crappy cheap software with lots of features fast and keep selling like crazy (Microsoft way), or you built expensive great code with only a fraction of the features (belive me, to add features to a given soft takes lots of time) and no single person will buy it.
!!Don't kill the messanger!! Kill the software buyers!!

Users can also be victims. They do not know what is best for them, partly because they are too uninterested for their own good, but mostly due to their lack of knowledge about software systems. Many of them have never known what quality software might offer, they have been made to believe that what is available is also what is possible:

Re:blah by **bmj** (#2255128)
we haven't reached the point when end users expect bug-free software. thanks to microsoft, users expect an application to have its quirks and problems. how many ms users have had windows 98 unexpectedly crash, and not even think twice about it? you just give it the three finger salute and wait for your box to reboot. there isn't a great deal of pressure on the development community (at least those who produce consumer-driven, non-critical apps) to produce *perfect* products.

The message of the moralising discourse in programming is quite straightforward: the right thing to do is to write high quality software, this is what 'real' programmers do. Economic and other kinds of external constraints may make this impossible, but it is still what *should* be done. Exactly in what terms quality is to be measured is not as clear, and definitely not as important. Sometimes there is no indication, the programmers just use the word "quality" (see *tshak's* comment above) and the conclusion I draw is that their point is that programmers, managers and users alike should (and usually fail to) show the respect that software development deserves.

So even it is not very clear what 'doing things properly' exactly means, this is not really a problem, since what is at stake is not the description of the right methodology but of the right attitude. Slamming code together to meet the deadlines is not the right attitude, demanding the resources needed to carry out a development project properly, on the other hand, is. Some participants had some more extreme ideas:

Accountability in Sofware Engineering by **JohnsonWax** (#2253786)
Last year I was talking with some high-ups in Boeing (VPs perhaps, I forget) about the need for licensing software engineers as Texas had recently begun doing, wondering what they thought of that move. While they agreed that there was need for accountability for software engineers (IIRC, these guys were planning YA-air traffic control net-

work) their argument against licensing was that there were no defacto accepted standards for code. That is, it's obvious to license a structural engineer - there are building, seismic, etc. codes to adhere to that have been written down.
Software has no such animals. No state (these are all state labor board issues) has ever written down that you should free a memory block after you're done with it, or check a pointer to see if it's null, and so on. Sure, these are accepted practices, but they aren't requirements.
As to previous posters that suggest that buildings are chaotically built, they seem to be overlooking tiers of state and local building codes, building permits, inspectors, plus the need for contractors to be licensed in most states. There's a lot of checking and balancing to be done and if it's wrong... Well, in the case of buildings that engineers need to sign off on (3 stories and up + special purpose, in most states) structural failure can result in criminal malpractice suits and jail time. If people die and the engineer overlooked something, possibly manslaughter.
Next time you are writing code, consider how you would approach the project and your boss if the prospect existed of the code causing bodily harm and you could be sent to jail? What if your code is used to control a traffic light, a power grid, an anti-lock brake system, an EKG display?

What *JohnsonWax,* and others, are suggesting is quite remarkable: they would like to see themselves being liable for the code they write. Together with their managers and their colleagues, naturally. This is moralising becoming law, and it is not forced from the outside, as one might have thought. Rehn (Rehn 2001) has described a similar process among warez doodz (warezonians), software pirates that crack comercial software and give it away for free. Even if their activities are illegal, they are governed by a set of rules created by themselves, which they try to enforce as strictly as possible. In both cases we witness the first steps, which may never come to a conclusion, of moral principles solidifying into regulations (Elias 2000) (Huizinga 1955). One would expect users to require these laws but instead we see that programmers, are the ones that, through their moralising discourses, suggest their necessity. Comments such as this one speak of the frustration experienced by programmers: they hope that by holding the companies responsible for the software they sell, programmers will be given the time needed to write high quality software, to have the possibilities that other kinds of engineers have. They also speak of their moral indignation: no-one shows the respect that software production deserves.

The moralising discourse that takes place in the Software Aesthetics discussion on Slashdot, and in the programming community in general, is more complex than the previous paragraphs suggest. There are no simple 'bad' and 'good' people, even it makes sense to introduce it in this way. The complexity is not only due to the rich diversity of programmers that take part in the discussions, but also, among other things, to the fact that programmers are not at all interested in constructing a comprehensive moral system. The moralising arguments and counter-arguments that they throw at each other are only the by-product of their real interest: programming, and talking / bragging / making fun about it. But it seems clear that what we see here is a community drawing boundaries as to who is a member and who isn't.

In this and the previous chapter, we have seen that some programming phenomena can be interpreted as two basic religious elements: rituals and sacrifices. Is there a point in comparing the private aspects of programming with a religion? In other words, is it fruitful to apply a religious metaphor to programming?

41 I am using a rather narrow, although dominant, sense of 'economic reason', namely its capitalist-utilitarian version. Bataille is slightly more careful since he makes the distinction between 'restricted economy' (my 'economy') and 'general economy', which includes a wider repertoire of exchanges (cf the following note). Alf Rehn, for instance, discusses the narrowness of 'economy' in the capitalist-utilitarian sense in his *Potlatch (*Rehn 2001), in which he describes a generous sort of software pirates (warezonians, warez doodz).
Sahlins notion of economy as the "process of provisioning society" (Sahlins 1972:185) is, I believe a much more comprehensive description of the economic phenomenon, but I have nevertheless decided to use 'economic' in the aforementioned narrow sense because it seems to me that it makes the reading easier and it is anyway unlikely to lead to misunderstandings. Besides, this is the notion of economy that programmers use
42 Bataille also insists in the notion of general economy and the essential role that sacrifice plays in it. General economy is a notion that Bataille opposes to 'restricted economy', which is the conventional concept of economy: the production, exchange and consumption of goods *among humans*. Bataille considers this view 'restricted' and insists that this movement of goods and services is only a tiny part of the whole movement of energy "on the planet's surface". The fundamental characteristic of this general economy is the generosity of the sun, which continuously provides with free energy. This, however, creates an imbalance which must be attended to. The only instrument that humans have to restore the balance is the sacrifice: only by means of an unproductive waste of resources can we maintain the equilibrium. His claim goes even further: unproductive waste – destruction of goods – will happen regardless of what we do, if we do not voluntarily destroy part of the surplus, powerful (general) economic mechanisms will take care of that – for instance in the form of wars. Perhaps we may assume that the atrocities of WWII had an influence in his thoughts (Hegarty 2000)
43 Am I perhaps using the word 'sacrifice' for a too broad set of activities? It is hard to know, but I admit that labelling any uneconomic expenditure as 'sacrifice' may be somewhat radical. On the other hand, this is exactly what I am after: inspired by Bataille, I want to distinguish between the investment and the waste, and signify the importance of uneconomic behaviour in activities as 'dry' and 'technological' as programming. As Sahlins argues in his seminal *Stone Age Economics*, economy includes production, exchanges and consumption unconcerned by profit (Sahlins 1972)

x
Programming as Symbolic Action

It is time to draw some conclusions, and in this chapter the argument is reviewed as an attempt at presenting programming as symbolic action. Writing a program is not only solving a computational problem (constructing a virtual machine that carries the intended function), it is also a process by which programmers create, reaffirm and communicate their world view and their place in it (through aesthetic preferences and instrumental beliefs, for instance); it is a way of expressing oneself. In this sense, choosing emacs instead of vi (see chapter 5) is not (only) the result of rational considerations but (also) a symbol of one's identity as a programmer. Writing software looks, from this perspective, more like practising a religion than like calculating.

We have seen how computing problems can be solved in a variety of ways, with different styles, and how the programmer can (must) make personal, i.e. non-calculable, choices. This characteristic of programming (which is by no means unique to it) makes of it something more than just an instrumental activity, namely a symbolic activity.

It is incorrect to think of instrumental actions as opposed to symbolic actions, these are simply two different perspectives on an action (there are others, such as the moral perspective). This does not mean that the study of *any* action from a symbolic perspective is going to be fruitful. For instance, putting some water to boil in order to make pasta and eat lunch needs not carry any symbolic meaning, in which case we may say it is a purely instrumental action. When I say that programming is also a symbolic action I mean that it also can be fruitfully studied from a symbolic perspective. In other words, that code, as the result of programming, carries a symbolic meaning: it is a symbol of, for instance, its author's aesthetic preferences and instrumental beliefs.

From an instrumental perspective, programming is the manipulation of commands (which are indeed symbols, but of a mechanical nature) in order to make the computer carry out the desired calculations. From a symbolic perspective, programming is the manipulation of symbols in order to create and communicate one's programming identity. This is, as we saw in the introduction, the aspect of programming that is ignored when it is equated with 'solving computing problems'.

In the previous chapters we have come in contact with different symbolic possibilities. Some of them were fairly simplistic, particularly those used to introduce the subject in the chapter on *coding styles* (variable names, commenting); some others were richer in nuance, such as the different aesthetic ideals and the question of prior design. In this chapter I shall round up this discussion, presenting one more alternative open for programmers to express themselves, namely the programming languages. The idea is not to go into each program's technical details but to present how programmers relate to the choice of a programming language. We shall see that, even if they explain this choice with technical details, what really makes them prefer one language to another cannot be justified only with technical data.

Choosing a programming language is only a very coarse way of expressing oneself, since it does not allow for much nuance, on the one hand because there are a limited number of options and on the other because any programming language allows many different kinds of programming styles. However, it is a choice that gives the program a general feeling, and working with C is not really quite the same as working with Pascal, let alone with assembler. Besides, each programming language has its history and its reputation: certain people use certain languages, and would dislike working with other ones. The intensity of these feelings varies of course from programmer to programmer, but, once again, we are not pursuing a comprehensive classification but an example of what the personal relationship between programmers and code looks like.

At any rate, programming languages are not only tools but also symbols, and programmers can use the choice of language to express their identity, making of it a decision in which more than just technical details come into play. But let us first have a look at how programmers on Slashdot discuss programming languages.

THE ISSUE OF PROGRAMMING LANGUAGES

Programming languages are one of the most important tools when writing software. As in the case with normal speaking languages, the commands (words, more or less) available and the kind of grammar that it is based on strongly influence what can be done and how. Consequently, the choice of language has a deep impact on the programming project: the look of the code, the sort of documents that can be produced, the speed with which some elements might be coded, the control over the actual actions of the processor, the type of access to databases, to other computers in the network, the possibility of programming parallel processes and many more.

At the same time, languages (at least the most common ones) are similar enough to pose problems when deciding which one to choose. There are not many programs that can be written in C but not in C++, in Pascal, in Fortran or Modula 2 (or 3) even if each language implies a particular approach. For the untrained eye it might be difficult to see the differences between code written in, for instance, Pascal and Visual Basic; but it might also be difficult for programmers working in a hurry. Connell's article, used as a reference in one of the Slashdot discussions, contains some code snippets, examples of “superior” and “inferior” code...

Re: If you ask me... by **Eric E. Coe**
[...] what I noticed was that the "Good" code was in well structured Pascal and the "Bad" code was is badly structured Basic. Do I detect a built-in bias??

Re: If you ask me... by **Anonymous Coward**
No, it just means that you aren't familiar enough with Pascal to notice that the article contains no Pascal code.

Re: If you ask me... by **Eric E. Coe**
My bad! You're right, I looked much too quickly at the first one, saw the 'const' at the top, and said "Pascal" - if I had looked more closely, I would have noticed the lack of semicolons, and the fact that the second example uses 'const' also.
Not familiar with Pascal? No. While it's been years since I've used Pascal, and I never liked it, the real problem was that I am not familiar with recent versions of Basic, Visual or otherwise (I stopped paying attention back in the GWBasic days).

Apart from the interesting detail that *Eric E. Coe* mistook Visual Basic for Pascal, we can also notice how he "never liked" Pascal, not because it could not achieve what rival languages could but because he does not want to be associated with the particular identity represented by the Pascal programming language. The fact that programmers like or dislike a particular programming language is a quite extended phenomenon, and generally they try to avoid working with the ones they dislike. This is not always possible, as *SillyWiz* explains:

Re:What a complicated question! by **SillyWiz** (#2256022)
I once arrived at a (fortunately) short project to find the specification /consisted/ of:
1) The project will use Microsoft Access, because the client has already bought enough copies of that.
2) The project will cost UKP 2000 or less.
3) The developer will not speak to the end users: they're too busy doing real work.
4) Whatever gets delivered gets vetted by the client management. Before any money is paid.
I did about 2Ks worth of what we guessed would be useful software and bailed. The users were /intransigent/ about not changing their working practices (which consisted of /RETYPING/ Word documents with minor changes in each draft), utterly unable to make decisions, and utterly unable to understand that there are some things Access is not a good choice for.
I have no idea what the hell happened at the end, I'm just glad I didn't stick around to find out.
It's not the first time a project has been like that, it's just an extreme case. And yet all these users are surprised by the software that results...

SillyWiz was clearly not impressed with the choice of programming language[44] that was forced on the developers on that occasion. Access is a Microsoft product, which, as we have seen earlier, does not make it particularly popular in Slashdot. The discussion about *Software Aesthetics*, one of the two that we have been analysing more closely, contains several manifestations of this aversion, the one I have chosen to present here being perhaps the most obvious. In order to understand the messages that will follow, you need to know that the article written by Charles Connell (*Software Stinks!*) about the general lack of quality of software included a couple of examples. They were simple programs only meant to illustrate that the same function can be written in different ways:

It is possible to have two different versions of a software program that function in exactly the same way, and have the same internal design and construction from a technical perspective, but which are vastly different in their human readability. Consider these examples.

CODE FRAGMENT A

```
Const MIN_AGE = 0
Const MAX_AGE = 120
Const RETIREMENT_AGE = 65

Dim AgeString As String
Dim AgeNumber As Integer, YearsToRetirement As Integer

EnterAge:
AgeString = Inputbox$("Please enter your age.", "Age?", "")
AgeNumber = Cint(AgeString)

If AgeNumber < MIN_AGE Or AgeNumber > MAX_AGE Then
     Msgbox "Are you sure you entered the right age? It should be between "_
     & MIN_AGE & " and " & MAX_AGE & "."
     Goto EnterAge
End If

If AgeNumber < RETIREMENT_AGE Then
     YearsToRetirement = RETIREMENT_AGE - AgeNumber
Else
     YearsToRetirement = 0
End If
```

CODE FRAGMENT B

```
Const xyz=0
Dim x As String,A2 As Integer,Y As Integer
              Const m =120
Const A=65
L47: x=Inputbox$("Please enter your age.", "Age?", "")
A2=Cint(x)
     If A2<xyz Or A2>m  Then
Msgbox "Are you sure you entered the right age? It should be between " _
& xyz & " and " & m & "."
Goto L47
End If
If A2<A  Then
     Y= A-A2
        Else
Y=0
  End If
```

These two code fragments are perhaps a bit simplistic but to be fair one cannot offer an example of proper code within the scope of an article. It would take too much place (my examples in the section about coding styles were also quite unelaborated). Connell's point was just to urge all programmers to a quest for beautiful software, not to give a comprehensive illustration of what he was talking about. But it seems many readers stopped the reading the article when they got to the examples: they were written in Visual Basic, a Microsoft product. Besides, Connell's company's website, where the article was published (the discussion on Slashdot only included a few lines of the article, with a link to the article itself), was designed with Frontpage, another MS product. Some slashdotters were clearly not impressed:

Re: And the quality HTML award goes to... by **Anonymous Coward** (#2253181)
HTML ala frontpage, code in VB.. and he's telling us about lousy software, what an ass.

***ACK* VB!!** by **Anonymous Coward** (#2252854)
The article has code fragments. Just now I noticed it was VB!
Why not show C++, C, or Java? That way you can really show the difference of a bad written program compaired to a good written one...

stinking code by **spektr** (#2253494)
Software aesthetics? Just look at the crappy HTML-code of this article:

```
<p class="DefaultText" style="text-indent: 0.5in;
line-height: 150%; margin-left: 0in">
<span style="font-family:Wingdings">?
<span style="font:7.0pt &quot;Times New Roman&quot;">
&nbsp;&nbsp;&nbsp; &nbsp;&nbsp;&nbsp;
</span></span>Cooperation</p>
```

What should be a <ul> is emulated with CSS and windings 8-bit characters (bullets, I suppose - they don't display on my system, because I'm not using windows!). A Frontpage-Consultant confesses his secret love for goto's and teaches us software-aesthetics using VB-examples. Strange times.

And the quality HTML award goes to... by **Brazilian** (#2252846)
... Charles Connell, for creating more "lousy" software. Call me crazy, but I would think that if you wanted to rant about "lousy" software you'd have the presence of mind to write decent-enough HTML so that the character " didn't show up as ? and bullets didn't show up as the character Y.

Wonderful-Comparisons between software and bridges by **sudog** (#2253016) [...] Large software systems are so completely different from real-world systems that comparing them is silly. (And is that Visual Basic I see there to try to prove your case with?)

um, is it just me ... by **codecowboy** (#2253388)
... or does anyone else find it is funny that the article discusses software aesthetics using Visual Basic code examples ...

Did anyone notice... by **The Slashdolt** (#2253023)
That in the revision history that this is the 3rd version of this paper in almost 3 years?
So it takes him almost 3 years to write a 10 paragraph essay with some VB code mixed in, and he is telling us we need to do better? Nice example Mr. Author.

A better look by **Lumpish Scholar** (#2253026)
The cited article doesn't say anything profound. (I got particularly worried when he said, "global variables and GOTO statements ... may be exactly what the software needs to marry form with function," and when his example of beautiful software turned out to be a fragment of Visual Basic. "It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential pro-

grammers they are mentally mutilated beyond hope of regeneration." [virginia.edu] --said, tongue at most partly in cheek, by Edsger W. Dijkstra, in "How do we tell truths that might hurt?") [...]

Is VB such a lousy programming language that anything written in it must be bad? I doubt it, but this is not the point. What matters is that some programmers strongly dislike it and would chose not to listen to anyone that uses it to give examples. For them, using it is a sign of poor programming judgement, regardless of what one may be capable of writing with it. It is, furthermore, not only a sign of one's technical identity, it is also a sign of one's political convictions, particularly in an open source-focused environment like Slashdot.

Microsoft products are not the only languages that are attacked. And the issue of programming languages does not only concern young angry programmers who dislike monopolies. In the chapter about community we saw how participants in a software-teaching conference made ironic remarks about some programming languages. We also saw how senior respectable programmers, even those with professorships, have their preferences. Donald E. Knuth, previously introduced, has this to say about Pascal, which he used as the programming language of the "WEB system", a sort of programming environment:

> [...] I chose PASCAL as the programming language because it has received such widespread support from educational institutions all over the world; it is not my favorite language for system programming, but it has become a "second language" for so many programmers that it provides an exceptionally effective medium of communication. (Knuth 1983)

He is far more discrete than slashdotters but it seems quite clear that he does not like PASCAL. It would seem every programmer, regardless of age and position, has either a favourite language, or a most-hated language. *Lumpish Scholar* quotes Dijkstra (see above) – another venerable programmer – in one of the preceding comments about VB. Dijkstra wrote a short and light-hearted article about some of “truths that might hurt”, among which I have selected these (about languages):

FORTRAN --"the infantile disorder"--, by now nearly 20 years old, is hopelessly inadequate for whatever computer application you have in mind today: it is now too clumsy, too risky, and too expensive to use.
PL/I --"the fatal disease"-- belongs more to the problem set than to the solution set.
It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.
The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.
APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.

Since not everyone’s favourite language is the same, and, in some cases, one’s pet is another’s language of hatred, disputes inevitably follow. These can deal with languages in general or with some particular feature of them.

One of the more controversial programming elements is the GOTO statement, a command that makes the processor jump from one place of the program to another (the command tells the processor to Go To a particular line in the program). Its detractors, among them Dijkstra, complain that “The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program” (Dijkstra 1968), its adherents counter that GOTO is very functional and effective. Dijkstra’s quote can be found in an article titled *Go To Statements Considered Harmful*, and at one moment, the dispute was so generalised that it seems every programmer had an opinion on the matter. Including D. E. Knuth, who wrote an article titled *Structured Programming with go to Statements* with the following introduction (Knuth 1974b):

Before beginning a more technical discussion. I should confess that the title of this article was chosen primarily to generate attention. There are doubtless some readers who are convinced that abolition of go to statements is merely a fad. and they may see this title and think, "Aha! Knuth is rehabilitating the go to statement, and we can go back to our old ways of programming again." Another class of readers will see the heretical title and think, "When are die hards like Knuth going to get with it?" I hope that both classes of people will read on and discover that what I am really doing is striving for a reasonably well balanced viewpoint about the proper role of go to statements. I argue for the elimination of go to's in certain cases, and for their introduction in others.

The differences have not yet been settled, even if the disputes are not as heated, as the existence of websites like *The Conversation Forum for the GOTO Statement*[45] prove. While this website has nothing to do with Slashdot, we are familiar with the tone of the discussion, which is perfectly natural as this site is also a place where programmers meet to talk about programming. The following is an example of the kind of exchanges one can find there:

Subject: In some languages, such as BASIC & Qbasic...
Posted May 7, 2001 by **Sage**
GOTO is faster then calling a sub-routine. I dont see anything wrong with using GOTO, unless you over use it; but isn't that true for anything?

Subject: In some languages, such as BASIC & Qbasic...
Posted May 8, 2001 by **SchrEck Inc.**
HI Sage, the keyword in this context is... modularization. You couldn't do independent subroutines, callable from everywhere, with GOTO because you couldn't automatically return to the place of calling. GOTO and GOSUB are completely different concepts and normally not interchangeable. And on modern computers with modern development tools, the performance of certain constructs or program instructions doesn't really matter in 99 percent of your code.

Subject: In some languages, such as BASIC & Qbasic...
Posted May 15, 2001 by **Kodiak**
While the use of a GOTO statement can be useful at times; I've found using them to be like a marble tower toy that I had when I was a boy. You could insert a marble into it from different points and the gods only knew where it would come out. The precise point of exit actually

was based on how you stacked the layers of the toy. However, more often than not, the marble either dropped out somewhere I didn't expect or worse... got stuck inside and I'd have to take the whole thing apart. If you are going to use GOTO's in a program, I would suggest using interesting labels for your exit points. GOTO BORNEO, GOTO BOBSPLACE, or the classic GOTO (a Judeo-Christian place of eternal flaming punishment) add color to your program for those who come along later to maintain it, or to fix the mess you've made of your program through the overuse of GOTOs.

Subject: In some languages, such as BASIC & Qbasic...
Posted Aug 14, 2001 by **xyroth** (rash enough to tackle intelligence)
you seem to forget that almost all languages tend to end up with the goto statement in them somehow, due to its inherent usefullness.
if your language is extensible, you can easily cut down on the number of goto's you need (and if you don't believe me, try reading any large bbcbasic program)
if you can comment properly, most of those problems with ending up where you didn't expect to disappear as well.

There is clearly no agreement as to which way to go. As Knuth says in his above mentioned article, the solution lies in a “reasonably well balanced viewpoint about the proper role of go to statements.” This is an opinion also shared by Dijkstra. He is quoted in Knuth’s article: “Please don't fall into the trap of believing that I am terribly dogmatical about [the go to statement]. I have the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline!”

It is noteworthy that Knuth and Dijkstra, both of them known for carefully considering things before speaking about them, chose to use religious terms in their references to GOTO. For instance, while Knuth used the word ‘heretical’ in his citation a few lines earlier, Dijkstra opted for the words ‘dogmatical’ and ‘religion’. The observation that the use of religious terms is rather commonplace among programmers is something I will develop later on in the discussion.

Disputes deal also with languages in their totality, for instance, with Perl, a language which is often the subject of both hard attacks and faithful support. This exchange comes from a newsgroup created in order to debate matters that concern a programming contest that the company *Google* had set up. Here follow the postings of some participants that became embroiled in a debate[46] about which languages should be allowed.

From: Christopher R. Wedman
address that here. Although it blows my mind why you would use object orientation with a project like this anyway.
Perhaps Google is looking for developers who can code OO, and one's ability to write _good_ OO code can be (best?) observed by examining C++, Java, and Python source - Not some Perl hack! (I mean 'hack' in the best sense of the word.)
I for one would not look forward to judging a programming contest involving Perl, unless it was some sort of obfuscation contest. It's too easy to write awful code. I think limiting the choices to C++, Java and Python would make judging easier, and the entries more relevant/ appropriate for Googles purposes.

From: Solhell (solhell@bigfoot.com)
It is always funny to read comments about perl from people who doesn't know perl. [...]
1) [...] It is easier to make stupid mistakes such as infinite loops and meaningless iterations in C and same code will be 1000 times slower in C than perl.
2) Perl is faster than C in some cases (in most cases related to this contest), and C is faster in some cases [...] It is crap to say that prototype in perl and then code in C++. That's stupid assumption in most cases. C++ is not going to bring you any advantages in most cases.

From: GMK (madflythug@hotmail.com)
Perl the best HAH! its a script not a real service can you imagine google using perl? every single person who searches creates a instance of the script yeah thats real efficeint, COBOL is better at parsing text than perl why cant we use that?

From: Jonty (jt@iterunet.com)
Come all you google whackers chanting insanely "perl. perl. perl!" [...]

From: Joseph Ryan (ryan.311@osu.edu)
Oh, darn... thats right, Perl doesn't have OO. Man, thats a real drag...

Oh, what's that you say? That was back in Perl 4? Perl 5 has been out for over 8 years now? So Perl has had OO for over 8 years you say... wow, who'da thunk it? [...]

Discussions, as we see, are not about temperate displays of evidence. On the contrary, they are full of sarcastic commentaries. Obviously, there is something more at stake than just presenting technical reasons for one's choices of languages. This was most evident when slashdotters expressed their contempt for Microsoft's VB. It would seem that some of them must present themselves as having nothing to do with it. There is a telling posting by *Procrasti*:

Java is inefficient by **Procrasti** (#2253348)
Even Java works squarely against the goal of "efficient". Give me C++ any day.
I've done projects in C, VB (im not proud), C++ (yep MFC et al, 5 years) and Java (1.5 years now), and I question the statement that java isn't efficient.
The poster is clearly at pains to set himself apart from VB. There is a palpable sense from this that there is a lot to be said about the contempt for MS, its connections to moral values, to economic situations and to group behaviour. Also there is a very interesting study to be made of languages that fare better than VB: a genealogy of the languages' symbolic value, so to speak. However, I shall break these discussions here, not without first inviting the reader to look at the annexe *Java vs. C*, which features a longish but worthwhile discussion between proponents of Java and C.

VANITY AND HOLY WARS

The existence of choices that define one's (programming) identity makes of programming a symbolic activity and a personal matter. The openly offensive and defensive attitudes towards programming languages are manifestations of this. These attitudes can in some occasions escalate to what some programmers call 'holy wars', a phrase I shall return to. First, a few words on one little detail from conversations at Slashdot.

Although rather insignificant, it illustrates well how programmers can identify with a piece of code. In the *coding styles* chapter we saw how *quartz* had an “irresistible urge” to write one-liners like: “join(" ", map {ucfirst} split(/ /, shift))” and how he could not help himself[47]. *fishbowl* reacted to *quartz* suggestion that such a one-liner was unreadable: “But there's nothing in this example that should be a problem for even a beginning perl coder, in my opinion. You've used a common perl idiom in a very efficient, clear, understandable way”[48]. It might be a case of a bit of bragging, from both sides. *quartz* wants to show his/her abilities in writing one-liners and *fishbowl* that s/he has no problem responding to them… suggesting that even “a beginning perl coder” could. *fishbowl* indeed proposes an improvement to *quartz'* one-liner: “If I were maintaining your code, I'd probably do away with your use of the $_, or at least, explicitly use $_ instead of implying it.” *quartz'* answer to that suggestion is quite telling:

Re:Beautiful software by **quartz** (#2254206)
OK, OK, you got me. The $_ was actually there when I pasted the code from the Emacs window, but just before I submitted I decided to take it out for added effect, as I know many non-Perl coders have, um, strong opinions about implicit variables. *Vanity, I guess*. But hey, it does work with strict and -w!:) [my emphasis]

It is not usual to see vanity explicitly mentioned, but I think we may assume that such a feeling underlies many of the comments we have seen throughout the thesis. Another good example of that are the commentaries that Connell’s use of VB triggered.

The issue of ‘holy wars’ is definitely related to that of vanity, as are all manifestations of the private aspects of programming.

‘Holy war’ is an expression that programmers use sometimes to denote particularly virulent discussions about programming issues. The expression is in occasions changed to “religious war” and sometimes to just “war”. It is difficult to say what percentage of programmers actually use these expressions, but it seems clear that the great majority, if not all, of the English speaking programmers are familiar with them.

One of those stormy discussions concerns the use of the GOTO statement, which we dealt with in a previous section. This is how *TowelMaster* from *The Conversation Forum for the GOTO Statement* remembers the days when the dispute was at its peak:

Subject: The h2g2 association for the abolition of gotos...
Posted Aug 28, 2001 by **TowelMaster**(ACE OMFC member of SATS)
Hello SchrEck Inc.
Well as long as it's not a democratic process you can be the vice-president....
How about lots of stories about impossibly stupid ways in which gotos were used ? Or, if we may expand a bit, we could reenact the 1980-s *Structured Programming Wars*. I must admit that I am more into that than into Gotos because I learned the programming-trade in the early 80-ies. *When the wars were most fiercely fought...* And I am one of those b****ds who has never used a goto in my life. And in those days men were real men, women were real women, and nerdy-looking programmers on coffee and coke were real nerdy-looking programmers on coffee and coke. [....]
TM.
P.S. I always did like the jokes you could program with gotos. Like "goto the-loo" et al. I know, I know, I was young... [my emphasis]

Other disputes that attain the status of holy wars are those that concern operative systems, programming languages (of which we also have seen a few examples) and editors (which we dealt with in the chapter on *coding styles*). Holy wars are often about technical issues, and they are seldom, if ever, settled. For instance, some people hated the GOTO statement and others claimed that it was efficient. And there is no way to prove anyone wrong, these are just matters of personal preferences. Dijkstra went as far as mathematically proving that anything that could be written with a GOTO could also be written without it (Dijkstra 1968), but that, of course, does not prove anything about its convenience. As we saw earlier, Knuth later wrote an article defending the (careful) use of GOTO.

Holy wars may make it difficult for programmers to collaborate, especially when the disputes become more entrenched. Exactly how difficult it will be to collaborate is impossible to quantify, it is, of course, valid to assume that some programmers will set their personal preferences aside for the common good. The point is not whether they stick steadfastly to their preferences but the fact that they exist, and are strong enough to generate heated discussions. *rho* seems to have had some difficult experiences:

Re: complexity by **rho** (#2254113)
[...] However, if you get more than 2 programmers in a room, they'll end up in some stupid religious war over editors or indentation style (or, God forbid, operating systems). Why? Because a lot of programmers are arrogant (most without reason), and will refuse to compromise their own "standards".

Programming disputes that deal with matters of taste (and not with calculable problems) share some of the characteristics of religious wars… and most of the disputes are indeed matters of taste; since disagreements about issues that can be solved through calculations are generally easy to settle. Programmers, however, use 'holy wars' sparsely and mainly to characterise arguments that have become intractable: anyone presenting any opinion on the matter can count on a smaller explosion. This susceptibility can be exploited in programming fora by participants who, I imagine, enjoy watching the turmoil. In Slashdot, those entries are often labelled "flamebait"[49] by the moderators, and there is an option that allows registered users (free service) to filter them away.

At other times, participants themselves understand that their opinions might hurt and chose to avoid sparking off a holy battle. Participant *a!b!c!* writes a message in which s/he gives his/her opinion on the subject of comments in the code, finishing with the following sentences: "Before I get flamed, I should include the disclaimer that simpler is not always better. There are times when you did need to use a clever chunk in order to improve performance, then it might be worthwhile to explain in greater detail as to whats going on. But for the most part, keep it simple."[50] The following one is a good example too (remember that Perl is a rather controversial language):

Perl Obfuscation by **SlipJig** (#483248)
For an example of how NOT to code, why not check out some Perl Obfuscation? I think it's a great thing to look at, even though (or especially because) I really don't like Perl as a language (heresy! heresy! Don't flame me please, I've heard the arguments).

"heresy! heresy!"... Time to look at the last point: is it fruitful to analyse programming from a religion-metaphor?

THE RELIGION-METAPHOR

Any attempt to speak without speaking any particular language is not more hopeless than the attempt to have a religion that shall be no religion in particular.... Thus every living and healthy religion has a marked idiosyncrasy. Its power consists in its special and surprising message and in the bias which that revelation gives to life. The vistas it opens and the mysteries it propounds are another world to live in; and another world to live in – whether we expect ever to pass wholly over into it or no – is what we mean by having a religion. (Santayana quoted in (Geertz 1973) :112)

Programming is not a kind of religion, for a number of rather obvious reasons. Programmers, however, use expressions like "holy war", as we have seen, and they sometimes refer to someone's attitude towards a particular programming methodology as "evangelism" or "extremism". *SlipJig*, in the last quote, used "heresy! heresy!", and this sort of religious terms appear occasionally in their conversations. But such expressions are not used systematically and programmers do not seem to think of programming as a religion in any deeper sense. For instance, programmers express themselves often in terms of opinions, and opinions and religion do not really go well together[51]:

The aim, in my opinion, is for self-documenting code. This is impossible in assembler. It's very difficult in C. But once you migrate to C++, then if you have a good design and good names you need very few comments. Most short methods don't require any comments at all, and adding superfluous comments is worse IMHO[52] than having none - they get out of sync with the code. (#483318)

Still, I think it might be useful to consider programming from the perspective of a religion-metaphor, since this perspective forces us to concentrate on the private aspects of programming. We have described this private sphere, to call it something, throughout the thesis, and we have seen that part of their behaviour looks more religious than deductive. For instance, programmers who prefer Perl do so based on personal conviction, not as a result of objective calculations. This kind of personal convictions may originate in the programmers' education, in their professional experience, in the conversations with other programmers, at work, in their political opinions, etc.; at any rate it does not originate in scientific measurements and comparisons.

We have examined different kinds of disputes about programming issues and also the existence and nature of aesthetic preferences in software. At the heart of all these phenomena lie those personal convictions, which could perhaps also be called beliefs. Those beliefs are not the same as the instrumental beliefs that have been discussed in previous chapters. It is, so to speak, more elemental. It could be said that instrumental beliefs are a tangible manifestation of the primary personal convictions. But what exactly do I mean by "personal convictions"? Is there actually some special kind of feeling inside programmers? I have written this thesis with the firm determination of not straying too far from the empirical data, so I am not going to discuss internal feelings of programmers. I nevertheless get the impression that the argument has veered in that direction, so I may as well bring it up explicitly.

For instance, as a summary of previous descriptions, I think it plausible to say that some programmers *believe* in Perl. With this I mean not only that they hold the instrumental belief that applications written in Perl are better (more useful, for example), but also that they relate to that programming language – in the conversations with other programmers, in his/her choices and preferences – as if writing programs with Perl was the right thing to do. At any rate, the only thing they want to do.

This 'believing' comes in many versions, and is manifested in different ways. Also, programmers that "believe" in Perl do so with varying levels of intensity. From those who have just discovered it and think it is cool, to those who master it, who have a long experience with it, who would rather not use anything else… well, who love it. The focus in those personal convictions shifts from the instrumental kind of belief (Perl produces better software) to a personal relationship with the language (tool), in which the importance of instrumental considerations slightly fades away. Perl, for the true believers, forms part of their attitude towards programming, of their identities. Hence, their belief in Perl is not based so much on facts as on subjective preferences. In other words, it originates in and is reaffirmed through their choices and their attitudes and not through the wielding of concrete proofs. For instance, they might very well know that Java, for instance, is also a good programming language with very interesting applications, but this knowledge, as well as comparisons of technical characteristics, leaves them unmoved.

However, the question of instrumental goodness is always present in an activity as instrumental as programming, and some programmers use it regularly to legitimise their convictions. Their arguments are seldom, if ever, the result of scientific studies, even if they are sometimes used as if they were based on universally valid facts. The following exchange is an interesting case of instrumental defence and critique of XP, a programming methodology:

XP! by **Proud Geek** (#2252936)
That is why we have advanced software engineering techniques like eXtreme Programming. Through it's constant refactoring it makes sure that code is always the best it can be for the task at hand, and constantly improving.
The only reason that so much code is ugly is that most people do not know about and adopt XP. XP closely resembles the reality of Open Source programming in its implement-now mentality and constant addition of features. If everyone used XP, the software world would be a better place!

Re:XP! by **Lumpish Scholar** (#2253069)
The only reason that so much code is ugly is that most people do not know about and adopt XP.
Extreme Programming (still abbreviated XP, despite Microsoft's attempt to dilute the abbreviation) may have a lot to offer many software development projects. But Kent Beck and Ward Cunningham and Ron Jeffries were capable of writing beautiful software before XP was codified, and programmers in XP projects are capable of writing ugly software.
Refactoring backed by unit tests (two XP practices) can help reduce software entropy, and keep software from becoming ugly; granted. But XP extremism helps no one.

Re:XP! by **Anonymous Coward** (#2253185)
programmers in XP projects are capable of writing ugly software.
Yes, and XP works anyway.

oh, that sounds like by **Anonymous Coward** (#2253280)
>> *programmers in XP projects are capable of writing ugly software.*
> *Yes, and XP works anyway.*
hmm..
yes it does, no it doesnt. Yes it does, no it doesnt. Yes it does, no it doesnt. Yes it does, no it doesnt...

XP didn't work for us by **GlenRaphael** (#2254697)
[…] At a previous job at a company which shall remain nameless, we tried to adopt XP. Hired a couple guys who were into it. Sent all the programmers to official XP seminars. The plan was to do our next-generation server using Smalltalk, replacing an existing system written in C.
It didn't work. After many delays and much miscommunication it became clear it was going to take too long to produce a product solid enough to replace what we had, if that ever happened at all. Eventually the company abandoned the whole project and went back to the old ways of doing things, fired all the XP guys.
XP is not a panacea. It, too, can fail.

Re:XP didn't work for us by **acroyear** (#2255455)
[…] Also, there's enough missing in your description of this that would lead me to think that the company is blaming XP for a true LACK of design. [...] Since XP requires extreme levels of communication and feedback, and the company was not achieving communication adequately, then its easily concluded that the company never really used XP, and therefore XP is not to blame.

XP? by **Anonymous Brave Guy** (#2253862)
This is typical XP evangelism, and as usual, it's supported by precious few facts.
XP has its good points, certainly. However, it's not nearly as clever as it thinks it is. "Test first!" they claim. Where do these tests come from? The requirements, of course. What do they do? They lead you to implement code that systematically meets the requirements. S'funny, I coulda sworn that was what this "design" thingy was all about.
And no, the fact that they UseIrritatingStyle for their
LongWindedNamesForThings does not make them clever, either.
[…] So, by all means highlight the strengths of XP. But let's leave the "Everyone should adopt it, because it's great, so there!" out, OK?

What is the authenticity of these apparent convictions? Do programmers like *Proud Geek* and *acroyear* really believe that using XP results, everything else equal, in better software? Or are they just pretending? This is a difficult question, and one that I cannot answer. I imagine that there are all kinds of attitudes, from the cynical to the sincere one. Ervin Goffman brings this issue up in his study of identity performances (Goffman 1990), restraining himself from drawing any conclusions. Yes, some people act cynically and some others act sincerely (and everything in-between) but this is not the issue. The point, he says, is that they *act*. My point is similar: some programmers do believe that using XP results in better software and some others simply *say* so (they may instead believe, for instance, that the choice of formal methodology plays a minor role). The point is not whether they really believe in the instrumental superiority of XP, the point is that they *argue* as if they did, and that they will defend this superiority against those who do not agree with (or should we say believe in) it.

As mentioned earlier, arguing for the instrumental goodness of the preferred language, or methodology, is only a way of manifesting one's programming convictions. These can also be held without much instrumental legitimising and programmers may simply prefer one editor to another, or an operative system to another, or a coding style to another, regardless of its perceived influence in the final result. In interviews, some programmers maintain that they refuse to use vi, or to indent with spaces, without presenting instrumental arguments but simply because "it's ugly", or "it's wrong." Not using vi is a part of their front (Goffman 1990), and it is not subject to careful objective comparisons.

From this perspective, the discussions about programming rituals and sacrifices that we have seen in the previous chapters may be reconsidered. But I would not like to sound as if programming is actually something religious, the terms 'sacrifice' and 'rituals' are used only in a metaphorical way. For instance, I think that to suggest that certain aspects of programming are sacred confuses more than it clarifies. In fact, taking the religion-metaphor too far would create a problem: what is this 'religion' that we are comparing programming to? The concept of religion is, despite a voluminous body of literature on the subject (or should I say, due to), hardly much clearer than that of programming. Therefore, I use it only in a superficial sense, as I imagine the reader has understood. My goal is not a detailed comparison between two very complex sets of activities but to offer insights into the nature of programming.

For instance, I mean that programming can be understood as a religion in the superficial sense that, in both cases, there is little about convictions that can be settled by discussions. What is there to be disputed about between different religions? That one is wrong and the other right? Similarly, If I 'belief' in C, I may agree that Java is suitable enough language and that, in some occasions, it might even be an interesting alternative, I may even agree to work with it, but nothing beats C (see the annexe *Java vs. C*). The disputes are not about what is the best option in a particular case but about which is *the language of choice*, in general. Ultimately, they are discussions about things that cannot be compared, and writing software is the activity through which personal convictions are expressed and generated.

44 Strictly speaking, MS Access is not a programming language but a small database designing environment, but it can be considered as one for the purpose of this argument
45 http://www.bbc.co.uk/dna/h2g2/alabaster/A354629
46 All the entries in the example are taken from the same thread in the same newsgroup. The subject of the thread: "Google is only solving 10% of the problem" and the newsgroup's address: google.public.programming-contest
47 Slashdot message #2253179
48 Slashdot message #2253306
49 Flamebaits are, however, not only about programming issues, as we saw in Slashdot's description of the concept (see chapter on method).
50 Slashdot message #483208
51 Wittgenstein says: "This is partly why one would be reluctant to say: 'These people rigorously hold the opinion (or view) that there is a Last Judgement'. 'Opinion' sounds queer." (Wittgenstein 1966:57)
52 IMHO appears often in Slashdot and it stands for In My Humble Opinion

XI
Closing Reflections

Research in the management of software development, and research in programming in general, should not ignore the personal aspects of programming. Programming managers, in turn, should not deal with them as simply something to be suppressed. I hope this much is clear after reading the thesis; but, given the picture of programming we have witnessed, is there anything we can say about other instrumental activities? Notably, about management? And is there anything to be said about technology at large?

My main concern in writing this thesis has been to present a thick description and a careful analysis of some of the private aspects of programming, i.e. of the personal relationship that programmers establish with the code they write. Originally, this concern war driven by curiosity, but the goal slowly became to devise a conceptual toolkit that would adequately explain some programming phenomena. Hopefully, the concepts will also be useful for the identification and study of similar phenomena in other instrumental activities (further on in this chapter I consider this possibility more closely).

Many books and articles have been written about software development, both from the managerial and the technical perspective. In most of them programming is treated as an activity whose sole goal should be to create useful tools, regardless of their intrinsic qualities. For these authors, the essence of code seems to lie in it being the solution to computing problems. Their readings imply that programming is about calculating, optimising, recalculating, testing, and recalculating again, all without a shadow of personal implication. As I have said earlier on, there is, a priori, nothing wrong per se with such an attitude: the pursuit of efficient software management is a perfectly legitimate activity. My aim has been to demonstrate that there is an important aspect that must be taken into account *if one wants to understand programming*, not that managers are wrong and programmers right. If the reader got this impression, it is probably because of my focus on the programmers' voice. Other voices in the software industry, notably the managers', are absent from the thesis (after all, the point was to present an internal perspective to programming), and even though I have tried to keep some critical distance from the programmers' arguments, their own opinions sound more legitimate than the ones *they* assign to managers. The point of this thesis is, once again, that a view of code limited to its instrumental value cannot explain certain phenomena that are an important part of programming.

By describing the private aspects of programming, I hope to have shown in what ways code is more than its instrumental value. By doing this I am following the steps of previous ethnographic research in the work of engineers, even if the case of programming has hardly been dealt with. In any case, I am by no means the first to have noticed that the creation of technology is not a matter of pure calculation. For instance, Gideon Kunda (Kunda 1991) offers a thorough description of the impact of, should I say, corporate aspects in the work of engineers. His subjects – engineers of an (for anonymity reasons) unspecified kind, although related with the IT industry – are clearly aware that the effects of their decisions are not limited to the technical realm, but that they are connected in intricate ways with strategic, political and organisational factors of their company. Technological creation, from this perspective, is not only about designing and constructing artefacts but also about making statements about one's affiliation.

Instead of the corporate, I have focused on the personal aspects, i.e. on the impact that the programmer's relationship to her own creation has on the activity of programming. And I hope I have managed to convince the reader that this relationship is a phenomenon rich in nuances and, more importantly, that it has the crucial consequence of turning programming (technical) decisions, which 'ought' to be objective, into questions of personal expression. I say that programming decisions 'ought' to be objective because, as I said earlier on, most of the literature on the subject (from popular management series to articles in scientific journals) seems to assume that this is how it should be. Why those authors should assume such a position is open to speculation. Perhaps it is because programming is regarded as technology, hence applied Science, and they make some sort of connection between both of them. Maybe they think that, since software engineers receive comprehensive training in mathematics and physics, the process of writing code should obey the strict methodological path of Science. They seem to ignore that Feyerabend and the Science Studies movement have proved (as much as these things can be proved) that Science is not at all a straightforward process. I can only guess that they accept an ideal view of Science as the ultimate rational effort, the one area of human activity where logic and objectiveness rule. According to their rationale, programming should also be strictly logical, without personal implications.

Most likely, however, the assumption is more the result of a methodology than a conscious decision. The knowledge that those authors are after is of a particular kind, its purpose being to make software projects more plannable, more controllable, to make them run more efficiently. Personal considerations are, from this point of view, a nuisance. They write about how things ought to be (in order to achieve efficiency), not about how things are. The question this thesis asks is how realistic is their vision, considering the realities of programming.

Whatever the reason, the fact is that personal aspects of programming have been largely ignored. Although they have not been totally invisible, they pop up in some, otherwise quite normative, accounts of programmers' activity. They also appear when discussing other 'non-objective' aspects of engineering work. For instance, even if Kunda (Kunda 1991) focuses on the implications of corporate factors (internal political struggles, corporate culture, salaries, responsibilities) in the work of engineers, he also observes that the prevalent image of engineers defines the nature of their identification with work and the personal characteristics that accompany it. Technology and its aesthetics are said to be the main concern of the engineers, who are driven by a fascination with "neat things" or "bells and whistles" – challenging features to design, interesting problems, and sophisticated, state-of-the-art technology. "The prize for hard work," it is said, "is more hard work." If these qualities are not available in regular work and assigned projects, they can be sought in "midnight projects" – the illicit projects that dedicated engineers are said to take on their free time for the sheer interest or pleasure of work. (:39)

For Kunda, these "midnight projects" have organisational consequences, for me, they are the traces of a specific phenomenon: the engineers' (or programmers') personal relationship with their creations, their concern for their intrinsic value. Brooks' classical *The Mythical Man Month* (Brooks 1995) also contains allusions to the personal aspects of programming, an activity that "gratifies creative longings built deep within us and delights sensibilities we have in common with all men". But he too leaves them aside to focus on other, also important, aspects of software projects.

I wanted to pick that embryonic idea and develop it. Like Kunda, spend the better part of a year carrying out observations in situ, attending meetings, taking notes, driving people around, asking permission to interview engineers and managers, etc. In other words, to gather empirical data through participant observation. I started with some interviews with programmers, but before I managed to obtain proper access to a programming group, I came across those two on-line discussions at Slashdot and my search for empirical material was more or less over. In those discussions, plus the interviews, was everything I needed.

I wanted to show that many of the technical decisions that programmers must make in order to write applications cannot be calculated. This last phrase means in this case that there is no way to know which alternative is the right one, that, in fact, there is no right alternative to choose. All options are equally valid, and the programmer must simply choose one. The key to this thesis lies in the fact that these alternatives, even if they are equally valid from the user's perspective, are not equal to the programmer. Hence, using long variable names, for instance, is not the same as using short variable names, choosing one alternative or the other will not change the functionality of the application but it says something about the programmer.

This is a very important point, I believe, because it opens up the activity of programming to a world of nontechnical possibilities. Since code and function are not univocally related, the same function can be written in many different ways, giving rise to the possibility of self-expression, of personal identification with one's code. This creates in its turn a whole array of phenomena that are not usually related to technological efforts: vanity, holy wars, aesthetic ideals, beliefs, etc. A set of phenomena that had been largely ignored and to which this thesis is dedicated: they are the visible traces of that end that I wanted to develop.

So this is finally the essence of this thesis: I have been looking for the traces, analysing the bits and pieces and presenting them in an orderly manner. I have been taking short steps around a rather small spot (comprising mostly two on-line discussions), in retrospect it feels as if I have been going through the material with a finetooth comb. When I now look up to see where I landed, I see that there are a few things that may have been left behind.

Getting access as a researcher to an organisation is seldom an easy task. So I started with the easy part of the process, asking for interviews. But there was something missing in both the questions and the answers, something I could not quite put my finger on, but that became quite clear once I came upon those on-line discussions. The richness of these exchanges made me understand that it is impossible (or very difficult) to gain an understanding of the personal aspects of programming just by asking related questions. What seemed to work was instead the dynamics of mass-confrontation: programmers found then occasion to choose sides, to criticise other's opinions, to endorse them, to give examples, to tell about their own experiences, etc. But I still miss something in this thesis: a proper (as opposed to virtual) ethnographical study of a programming group. Not so much because this study is incomplete without it but because I really would have liked to do it. As it stands now, this will have to be studied in another project.

There are many positive aspects of virtual ethnographies (see the *methodology* chapter) but there is also one, at least, negative aspect: it never finishes. This it shares with normal ethnography, where the researcher finds herself taking notes of as much as possible (clothes, arguments, agreements, coffee-machines, contracts, greetings, etc.) but always missing things out. In her case, the problem is the obvious metaphysical impossibility of putting down life on a piece of paper; in my case, it is the (perhaps only material) impossibility of following all the links in the internet. Participants in the discussions make reference to other sites (personal homepages, corporate sites, other on-line fora, open source code, etc.), which in turn link to other places, which in turn… the on-going discussion about the personal aspects of programming condensed, so to speak, in those two Slashdot discussions, but they are only the smallest tip of the greatest iceberg. Programmers are engaged in a continuous exchange (not only on-line) that forms the real backbone of the programming community (chapter nine), and that influences the form of the relationship between them and their code. There are a hundred different aspects of this exchange that have not been dealt with here, mostly due to lack of space but also to allow for *some* kind of order. So this study may very well be completed with other studies that focus on other aspects and other forms of exchange among programmers.

EFFICIENT PROGRAMMING

This thesis argues that programming is an activity that does not obey a strict instrumental logic. Writing software is a creative process, involving programmers in personal ways. As we have seen throughout the book, these do not simply calculate the solution to computing problems, in fact, they perceive code as a creation that represents them.

Some observers must wonder if this has to be so, if programming cannot be reduced to an automatic – hence cheaper and more reliable – process. I do not think so. Granted, that does not mean that some phases of the process can be, and have already been, automatised. For instance, graphical programming environments allow programmers to create data structures by simply dragging and dropping elements into a window. The environment automatically transforms this window (some sort of blueprint) into code. Not all programmers enjoy these methods (which denies them total control of their code) but the number and variety of graphical environments on offer illustrates the concept's popularity.

However, the only thing those environments can do is to facilitate the coding. The code produced by them represents only the most elementary structure, the rest must be filled in by hand, so to speak. And what's more important, they create neither design nor specifications; the two fundamentals of any program are still the fruit of human creativity; they are truly manufactured (this is the essence of the art of programming) following thus a much richer rationality than strict logic. In other words, the creation of specifications and design require a full human approach.

It is certainly debatable whether this approach could be made, if not totally automatic, then at least more controllable. In other words, if it would not be possible to eliminate from the approach clearly unscientific aspects such as holy wars, aesthetic quests and other rituals and sacrifices. In fact, a programming methodology is a set of guidelines for improving efficiency. Such an improvement is, in most cases, to be brought about by closer control of the programming process. What exactly this 'closer control' involves varies a lot. In some cases it is based on a detailed decription of the phases of a project (so that progress can be better measured and monitored), in others on a near collaboration with the users (so that programmers do just write according to their own preferences). In all cases, the goal is to put the instrumental before the intrinsic value of code. Arguably, they are saying that even if programming might not consist of rational assessments, calculations and weighing of pros and cons, it certainly *should*. I assume that the reason why programming should be made more automatic is the belief that code written according to strict methodologies is not only cheaper but also better.

This, however, is only a belief. In an ideal world, everything fits together, there is ample time for everything and strict methodologies not only make sense but will most likely result in cheaper and better software. What are the characteristics of this ideal world? One where the final users are absolutely taken by the idea of using a software system, and are capable of analysing their everyday work so as to transform it into abstract data structures, and have the time to do it, and where programmers are ready to listen to the users, and have the patience of examining their jobs, and have the skills necessary to write the code, and get well along with each other, making collaboration swift; and where 'managers' do not make impossible delivery promises, and are more interested in creating the code needed by the customer than by their own careers.

But we only live in a real world, and strict methodologies that make programming more automatic and less human *may* not be the best solution. The only thing we know is that they are practically impossible to implement: humans are still only humans. If not even Science progresses according to strict methodologies, how can we expect such an order in the writing of a program, which must not only attend to the *regular* complexity of Nature but, in Brooks' words, also to "arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which [the software's] interfaces must conform." (Brooks 1995) (:184).

Certainly it could be argued that programming would be improved by eliminating one of the disturbing elements? Would programming projects not become more foreseeable if programmers stopped relating to their code in personal ways? After all, the phenomena presented in this book (aesthetic considerations, holy wars, vanity, etc.) do not have any positive effect on the productivity, do they? It seems that, in the best of cases they do not interfere with the swift progression of the programming project, right? Well, I imagine that these private phenomena have all kinds of effects on the swift progression of the project, in some cases they may incite the programmers to work harder, in others to sulk and drag, in some to big blunders, in others to great insights. But the discussion is merely hypothetical because the case is that these private phenomena cannot be eliminated. Programming is not a matter of calculation; regardless of the strictness of any methodology, programming is always, at its most fundamental level, a creative effort. And this effort is always a personal one, and will always give raise to issues such as those we have considered in this book.

Therefore, any effort to eliminate the personal aspects of programming will, in the best case, only have the effect of relocating them: if variable names must all have three letters, denying the programmers the possibility of using long_winded_expressive words then they will find other ways to put a personal touch on their variables. And if that is denied too, they will come up with something else. In the worst case, it will irritate programmers, who may not see the efficiency gains that those measures promise, and feel they are forced to follow stupid rules. Or even worse, they might perceive those rules as a proof of lack of respect for the art of programming, and work actively against them and anyone who tries to impose them.

As I said, my concern has been with describing and analysing the private aspects of programming, not with constructing a management methodology. As far as I can see, in fact, the only sensible recommendation to managers that can be extracted here is "Bear in mind that programmers see code as much more than just the solution to a computing problem." What exactly is to be done, i.e. how these words can be put into action, is a matter best left to the managers themselves, who are familiar with their particular circumstances. What I have done is to describe the manifestations of the private aspects of programming, so that anyone may recognise them; I have also analysed them, suggesting a way of interpreting them, so that anyone may not only recognise them but also understand their role in the context of creating software.

This thesis marks the boundary that delimits the validity of a rational assumption of programming. By showing the importance of the personal aspects of programming, it sets a limit on what can be expected from treating programming (both as a researcher and as a manager) as a rationally driven activity. An example may clarify this point.

In the chapter about coding styles I took up an article written by two scholars (Oman and Cook 1990) that dealt with the question of typography in code. They suggested a particular kind of layout (book format model) and proved, with the help of laboratory statistics, that this layout increased the efficiency of programmers (by allowing them to understand the code faster). One may assume that, view that "professional programmers can benefit from the book format model", their implicit recommendation is that programmers adopt it. But programmers, as we have seen, are not only worried about understanding code, or making their code understandable. What they seek is to write code that they can be proud of. Hence, they are more likely to adopt Knuth's methodology, even if not backed by any statistical proofs, simply because they admire Knuth as a programmer, and are glad to see themselves associated with his name. And will programmers, for instance, accept and adopt the bug-detection systems that The Economist speaks about (see the introduction)? Well, I hope it has become clear that the answer depends on a number of circumstances, only some of which have to do with reduced costs and less buggy software.

MANAGEMENT, INSTRUMENTAL ACTION AND THE RATIONAL NIGHTMARE

While programming has been the main focus of this thesis, we have also been looking at something broader: instrumental action. The concept has been touched upon in the thesis, when it was used to describe the public view of programming: an activity carried out in order to achieve a goal, not for its own worth. Now, there are a number of other instrumental activities, notably management, and the question here is whether the present work may throw some light upon them as well.

The main finding, if we can call it so, of this thesis is that programming has a value in itself, at least for those in charge of doing it, and that therefore, studies that treat it as if it were a perfectly objective activity cannot explain an important number of (clearly existing) phenomena. Programming is, therefore, not a purely instrumental action, and researchers should not reduce it to one. Now, does this also hold for 'management'?

Management has been treated here (caricaturised, one could say) from the perspective of programmers, who use that figure as a straw man that personalises indifference towards the intrinsic value of code. But is it possible to say that there exists such a thing as the intrinsic value of management? And that it plays a role in management decisions?

Naturally, if I needed a whole book to explain what the intrinsic values of programming are (and hence, what programming is), I am not going to be able to explain what management is in a few lines. Nevertheless, I think it is possible to outline the concept of the private aspects of managing, just to show how the ideas presented here can be applied to that field.

Perhaps managing can be described as the art of making things happen, or of getting things done. A manager is a person in charge of a process, her responsibility is to achieve a certain goal (for instance 'to increase shareholder value', or 'to produce a program within budget and schedule'). A perfectly instrumental view of management would hold that the means (management methodology) are not important, only the goal. In other words, that management methodologies have no intrinsic valule, they can only be examined according to the results they provide. Exactly what should be included in those 'results' is not easy to determine; for instance, it is unclear in what measure ethical considerations should be considered.

At any rate, the public (instrumental) view of management methodologies is that they are not implemented for their own sake but for the results they can provide. From a private perspective, on the other hand, management methodologies have intrinsic value. In the same way as a coding style spoke of a programmer, a management style should speak of a manager. Now, if coding styles manifested in the code (it is possible to read a programmers' style in her code), where do management styles manifest? And, if holy wars, vanity and aesthetic ideals were some of the private phenomena of programming, which are their management counterparts?

Answering these two questions would take a complete new thesis, but we can at least point to some similarities between programming and managing that suggest the existence of private aspects in the latter as well. For instance, we saw how the private aspects of programming originated partly in the fact that the same result (function) can be achieved with different codes. In order to minimise the technical overhead, we only looked at rather superficial examples – the possibilities offered by a few coding alternatives –, but the message was clear. If there is only one solution to a computing problem, choice is annulated, and with it, the possibility of expressing oneself with code.

The contrary is not *necessarily* true: we cannot logically deduce that the possibility of achieving the same result in several different ways implies the existence of private aspects. I can wash my hands in different ways and I still have not seen any evidence of private phenomena of washing hands. Perhaps there is a minimally-complex-activity threshold that must be surpassed before private aspects appear, but this is something that would have to be considered more carefully.

At any rate, management does offer choice: any given management problem can be solved (or avoided) in different ways. The question is whether these choices speak about the manager, and whether their intrinsic qualities (regardless of their result) are ever considered. And also whether this gives rise to aesthetic ideals of some sort or not.

Another aspect that both activities have in common is that there is as little to science of management as there is to science of programming. And, before I get flamed, I hasten to explain this statement. What I mean is that we know as little about the concrete results a certain management methodology will poduce as we know about the best way to solve a given computing problem. What formulas do you apply in order to ascertain an increase of share-holders value? There are none. There are strategies and there are methodologies and there are financial tricks but there is no general solution to this problem. Managers must make decisions that cannot be backed by a scientifically valid set of findings: should one focus on the core business (and what is that?) or diversify? increase margin or expand (and are they incompatible)? should one take more or less risks? This also holds for more concrete decisions: three of four departments? What kind of salary / holiday / pension / etc. policy? How many hierarchical levels? No-one knows categorically, the best thing one can say is that "it depends", but on what exactly it depends is not so easy to specify. We still do not have a working science of management comparable to the working science of wave propagation. This lack of reliable formulas (lack of reliable forecasting) gave raise, in the case of programming, to instrumental beliefs: some programmers believe it is better to do so and so, and some other programmers believe the contrary. Instrumental beliefs exist also in management (and are sometimes called intuition), and strengthen the case for the existence of private aspects.

It would seem that the personal choices that managers face are wide and important enough to assume that the private aspects of managing exist and play a role in it. There is a lot of research that shows that management is an activity rich in aspects and nuances and that it is extremely unrewarding to think of it as a calculating process (this is indeed commonplace nowadays). Managers do not base their work on the computation of the optimal way to achieve their goal (if there is one clearly specified at all). The question is whether one can find phenomena that can be best explained by assuming the existence of an intrinsic value of management methodologies in the same way as we have here assumed the existence of an intrinsic value of code.

Furthermore, one must find the places where such an intrinsic value can be constituted, i.e. the places where it can be discussed. Slashdot is a great example of a place where the intrinsic value of code is constituted, and it is not the only one. Probably, the origin of such a concept is to be found in the classrooms where programming is taught… but that is another story. I doubt the internet will prove as fruitful in the case of management as it was for programming, perhaps one should try MBA schools instead. Existing research on managers, particularly that with an ethnographical approach, may also offer interesting observations: what do managers talk about when they are not making deals, when they simply chat about what it is to manage?

Last but not least, one must also find the objects through which the managers' express themselves. In the case of programming, the most obvious example is the code. Code not only is the immediate object where their coding styles are articulated but also the place where one can read design strategies, instrumental beliefs, skills, etc. Code speaks about its creator. Now what speaks about a manager? The organisational structure? The formulations of strategies? The human resource policies? The mergers? I am certain relevant phenomena have already been unearthed, it may only be a matter of ordering the material in this fashion. I think it will prove useful to explain it.

The question of the private aspects of programming (and of management) is an essential part of the abstract concept of instrumental action. And in this case, with 'instrumental action' I mean a negative kind of rationality that has been blamed for the reduction of the human condition. From Heidegger (and other earlier critics of technology such as Mumford (Mumford 1952) and Ellul (Ellul 1964)) to the contemporary (and less known) Higgs, Light & Strong (Higgs, et al. 2000) (see also (Feenberg 1999) and (Mitcham 1989)), the last century has seen a number of critiques of the reducing effects of approaching life with an instrumental attitude, sometimes known as 'technological mode of thinking.'

Certainly, due to its economic power and its strive for standardisation, technological thinking seems to have an upsetting reductive effect on human nature. The humanist perspective identifies the main reducing mechanism of technical thinking in its quest for efficiency, which is based on a view of the world as raw material, ready at hand to be *optimised*. This effort of constant optimisation and, particularly, the idea that this optimisation is possible and *desirable* (Wright 1994; Wright 2000), are identified as the main forces shaping our destiny. This technological effort and conviction go hand in hand with its scientific counterparts, whose goal is to take off the veil that covers the mechanisms of nature, and to express these in mathematical formulas. If this objectifying view of the world was not bleak enough, we have to add to it the economic conditions in which it exists (and thrives). The general background of human existence created by these two forces (the economical and the technological) make some humanists rather pessimistic about the future.

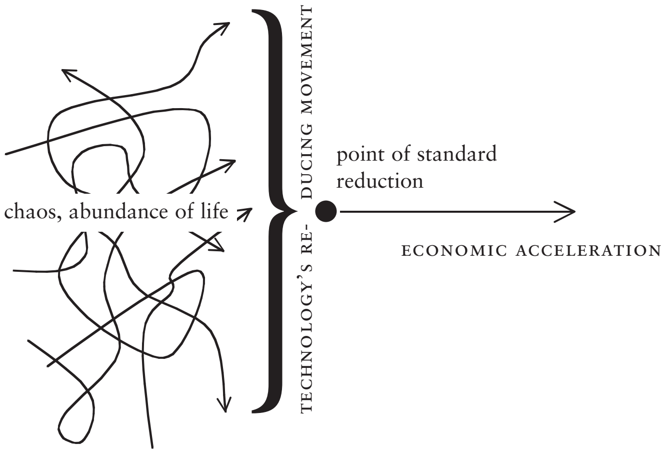
I am also appalled by some of the outcomes of this atmosphere of greed and technological possibilities, and I too worry about nuclear disasters. But, and this is my contribution to this debate, the technological mode of thinking cannot be limited to a view of the world as raw material, and more concretely of technological artefacts as cold objects of optimisation. This is the view of the world that results from (or in) the traditional philosophy of science and its strict methodology, but neither the scientific effort nor the technological development follow any such reductive paths. As Feyerabend has noted (Feyerabend 1987), (natural) sciences 'advance' in anarchic ways, and the technological development is, if possible, even more disordered (as we have seen here and as can be seen in the work of Kunda, Latour, and in works contributing to the STS movement).

Guillet de Monthoux' work *Dr Kant* (Guillet de Monthoux 1981), that studies a particular kind of technological thinking (the one found in technical standardisation committees) may serve to illustrate the point. His is a sharp and critical work that exposes the artificial nature of standards, the incorrectness of the economic assumptions that lie behind their adoption and the reductive effect they have on human creation. Guillet de Monthoux studies the work of standardisation committees: their configuration, their organisation, their agendas, their composition, the success of their conclusions, etc. coming up with an image of the standardisation process as a rather disordered procedure whose legitimising discourse is full of logical holes. Standards, he observes, are not created according to a well defined methodology but instead constructed in the corridors, in-between meetings and around coffee machines. So, Guillet de Monthoux detects the phenomenon that we have studied here, but stops in front of it, being more interested in a reflection on the normative quality of standards than in the nature of the deliberations held by the members of the committees. According to the findings of this thesis, one would expect these deliberations to involve all kinds of human faculties, not only economic, mechanical and political rationality; and one would expect to find traces of the phenomena associated with those ‘other’ human faculties (pride, aesthetic pleasure, vanity, fantasy, disdain). These would not invalidate Guillet de Montoux’ critique of the artificial nature of standards, or of their economic assumptions, but they might rebuff the general notion that the creation of standards is a process devoid of any form of intrinsic qualities, that it does not allow personal expression.

But more important than the process of creation of standards is the idea that standards, themselves, reduce the possibilities of human action. On one hand, this reduction is a clear effect of standards: once they have been accepted, and adopted, you cannot simply start constructing artefacts that do not comply, either because it is illegal or because it is not, a priori, economically sound. In this sense, it is also clear that the existence of standards may make life easier for the consumer. Both these ideas are clearly stated in *Dr Kant*, and are both applicable, on a certain level, to programming: every programming language can be considered as a set of standards that define what can be done with the microprocessor (itself standardised). But this is where *Dr Kant* stops, and where I began: in the study of how standards are actually used, or, in other words, how human creativity and fantasy flourish even in the most strictly standardised of environments (computers and programming languages).

As we have seen in the previous pages, when we study the private aspects of programming we discover how the creation of software engages all kinds of human faculties. It is a mistake to reduce the technological effort to a quest for the most efficient solution, unless one is ready to widen the concept 'efficiency' beyond recognition. In this thesis we have seen programmers relating to software, both their own and others', in ways that cannot really be described as 'optimising', and also how the theoretically fearsome 'efficiency' loses much of its edge when it comes in contact with, as Feyerabend would say, the abundance of life.

From close up, programmers seem unable to agree on what 'efficiency' means in every concrete case. Yes, they all know that it is a measure of output to input but there are diverging opinions as to what the input and output are, and as to how they should be measured. Instead of maintaining an emotionless optimising attitude to the world (object), they end up having quarrels about which are the most beautiful programs, which are the best programming languages and who is entitled to have a say in those discussions. In other words, the pessimistic approach to instrumental actions (rational attitude to life) assumes a condition, illustrated in the figure below, that does not exist:

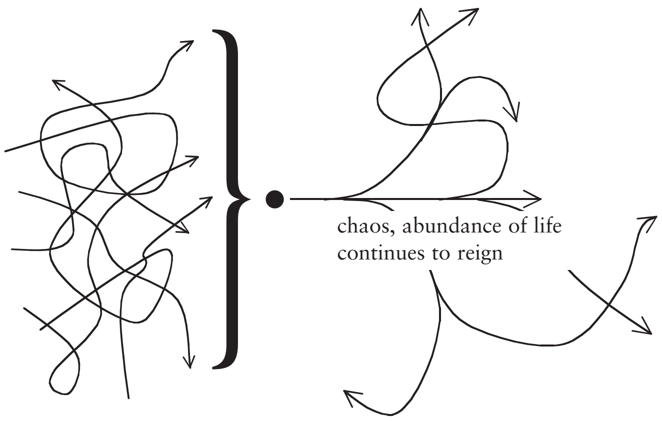


FIGURE, The Rational Nightmare

The disorder to the left represents the richness of human activity in absence of the technological mode of thinking. When this latter takes over, though, it strives towards one point of standard reduction, of efficient production. Here our possibilities have been limited by standards and, above all, by the focus on the result of the action, instead of on the action itself. The limitation is made more restrictive by the power of economic efficiency, which seems to accelerate the whole movement.

Now, do not misunderstand me, this *is* happening, in a way. Standards, as Guillet de Monthoux shows, do appear. Their very goal is to put limits to our alternatives of action (of creation), and their profitability makes them extremely successful. And this double efficiency (technological and economic) has given them an influential role in our society.

What this thesis shows, however, is that our inventiveness knows no limits. At any rate, the creation of standards does not limit our *practical* freedom. In a world as strictly standardised as that of programming (in which computers do not accept the least deviation from a predefined vocabulary and grammar) we find heated disputes about the best way to create. Instead of an accelerating straight path towards the optimal solution we find programmers taking all kinds of deviations, and this does not happen mainly as a rebellious reaction but as the necessary result of the abundance of life. There simply isn't a unique optimal solution, the strict standard rules of programming languages forms the base from which to create a new world, as the following figure illustrates:



FIGURE, The Abundance of Life

To sum up, standards organise and put limits to the situation from which they arise (represented in the figure with the point of standard reduction), but to the subsequent situations (the future), they are simply the points of departure for renewed chaos. The situation is well known for all of us that have had to learn to use personal computers: a word processor creates standards of many different kinds, reducing the alternatives of action but, from the other side, so to speak, it is the platform from which our inventiveness is re-launched.

Furthermore, the chaos of human industry cannot be bridled by applying strict methodologies – as the aforementioned *Dr Kant* shows – on the contrary, only the anarchy of inventiveness, forgetfulness, pride, stupidity and other human resources can bring about new points of standard reduction (Feyerabend 1978; Feyerabend 1987). The *theoretical* description of a technological mode of thinking may be devoid of humanity but, as the philosopher Ortega y Gasset may have put it (Ortega y Gasset 1996), the *real* technological effort is a very human enterprise indeed.

I
Annexe

JAVA VS. C

Programming languages are a dear subject to many programmers. They do not appreciate their preferred language being criticised and there are a few instructive and entertaining exchanges about this. Excerpts from this appear in the text, but it is interesting enough to be transcribed in its entirety in this annexe. It deals with the different excellencies of C and Java. It starts almost by accident: participant *Telek* sends in an entry in which he/she blames economic thinking ("once you have money entering the picture, and/or time, then the first thing to go is code quality") for the existence of so much poor quality software ("I can't tell you how many software packages I've looked at that are ABSOLUTELY HIDEOUS on the inside")[53] and rants mostly against Microsoft's Visual Basic but manages to comment upon Java as well:

And How!!! by **Telek** (SW)
[....] Unfortunately, in the land of "80% complete is good enough" and where "as long as it works" is a good philosophy, and in a land where "visual basic" is a professional programming language, we're not going to see this improve any time soon. Even Java works squarely against the goal of "efficient". Give me C++ any day.

That mention of Java as a, what should we say, perhaps-not-so-great language sets off a thread (*Java is inefficient*) of 10 postings of which I have selected the following. Please watch how even if they fence with technical arguments, and they do bring up a few, they do not leave matters settled: they finish without final objective conclusions. You would say they are just working on the building, and expression, of their identity.

Java is inefficient by **Procrasti**
> *Even Java works squarely against the goal of "efficient". Give me C++ any day.*
I've done projects in C, VB (im not proud), C++ (yep MFC et al, 5 years) and Java (1.5 years now), and I question the statement that java isn't efficient.I guess the gripe I have with this statement is your

definition of efficiency. I won't argue that Java executes slower than an equivelent C++ program, after all it runs on a virtual machine that does have to do an amount of work to translate java byte code to native executable code, however, Moore's law applies here - machines and JIT (Just in Time) compilers will always get faster, as well as implementations of Swing and other graphics libraries.However, I also like to think of efficiency in terms of developer hours, support and maintainence. Java outshines C++ in its ability to clearly express your ideas in a way a machine can understand. It frees the programmer and maintainer of the details of memory management. Not that a developer doesn't have to understand memory management and the implications of holding references to objects, but that allocation and freeing of memory isn't a constant requirement. Every C++ program of sufficient complexity has to be tested and debugged for memory leaks - someone always forgets to add a destroy call somewhere. I won't even mention buffer overflow problems While I'm on the subject, and although I seem to be praising Java a lot here, there are always places for each language. I don't think Java would be a good choice of language to build a kernel in, for example. [...] To sum up, Java is generally a more elegent language than C++, this leads to code with quicker times to market, less bugs and less cost in support and maintenance - efficiency isn't everything, afterall, "premature optimisation is the root of all evil" -- Donald Knuth, and how much more premature can you get than in choosing the implementation language?

Re: Java is inefficient by **Telek**

> *I guess the gripe I have with this statement is your definition of efficiency*

Fair enough. I meant fast. And Javascript is even worse.

> *Java outshines C++ in its ability to clearly express your ideas in a way a machine can understand*

How so, more than C++ OOP? Java is so very close to C++ OOP that I'm not sure what you mean.I will agree that Java has a lot of ADD IN LIBRARIES that come standard, however this is part of the packaging, not the language.I want to make a linked list in Java. Ooops, no pointers, sorry. I want to pass a variable to a function and have it modify it, oops, no pointers. I want to write a program that takes as little memory as possible, or reuse memory, or optimize it to use common options of the processor, oops, no memory management, no assembler. I would really like to see a pseudo assembler in Java, I think that'd be kickass. You can write a platform independant assembler (I did for my last job, well, it translated into a few different assembly languages).
Java gets ease where it says "uhh, no, shut up, sit down, I'll do that for you (and BTW, you can't do that".

> *allocation and freeing of memory isn't a constant requirement*

I never did understand this gripe. Whenever I put a malloc, I immediately put a free. Whenever I do a new, I immediately put the destroy

somewhere. And there are umpteen packages that do both source-code level and runtime level checking for memory problems.
> *I won't even mention buffer overflow problems*
In light of the recent IIS problems, I don't blame you. However there are packages that can scan for and test these things. And those problems are generally the result of sloppy programming in the first place, but I guess that's what the whole article is about. Yes, however, it is nice to know that you can't have that happen in a Java program.
> *there are always places for each language*
Absolutely. Java is easy, quick and dirty. You can RAD things with great ease. If development time is an issue, and you don't have the needed libraries in C, then Java is a great language. Just as long as you don't need speed or to get to the nuts-and-bolts of things. [...] Damnit, I want a programming language that gives me access to the freeking carry flag! =). I've done math routines a lot, and the code is literally 10x faster when you can optimize it by hand in assembly. I love assembly for small things that you want speed for. Itanium assembly is amazing for that task too (but much much more complicated). I guess that my gripe is because I'm coming mainly from a C background where you can do things like memory management, pointer management, and inline assembly. I am a big one for code efficiency and speed. When you're writing something that needs to run fast (i.e. a server), it pains me great to hear the execs go "well, we are going to have to buy some more servers so other expenditures are going to have to be curtailed a bit", knowing full well that if they weren't running on Java, or if they gave us the time to optimize the code base, you could run everything we have on half the amount of hardware that we have currently.

Re: Java is inefficient by **Turing Machine**
> *I want to make a linked list in Java. Ooops, no pointers, sorry.*
So? It's trivial to code a linked list in Java. The "no pointers" FUD is just that, FUD. References to objects in Java can be used for just about anything you'd use a pointer for in C++.

Re: Java is inefficient by **Procrasti**
> *I want to make a linked list in Java. Ooops, no pointers, sorry. I want to pass a variable to a function and have it modify it, oops, no pointers. I want to write a program that takes as little memory as possible, or reuse memory, or optimize it to use common options of the processor, oops, no memory management, no assembler. I would really like to see a pseudo assembler in Java, I think that'd be kickass. You can write a platform independant assembler (I did for my last job, well, it translated into a few different assembly languages).*
Well, everything in Java is passed by reference. From a C++ programmer's point of view, rather than thinking, Java has no pointers, its best to think, everything is a pointer. Its far easier to program a linked list

class in Java than it is in C++. I've done both. If you pass an object to a method, that method can modify the object.
The only thing that it won't do as well is type safe linked lists, because Java has no templates, but have you ever taken a C++ template and tried to compile it in VC++, Borland C++ and gcc, #ifdef everywhere to get this to work.
> *I never did understand this gripe. Whenever I put a malloc, I immediately put a free. Whenever I do a new, I immediately put the destroy somewhere. And there are umpteen packages that do both source-code level and runtime level checking for memory problems.*
Yeah, except when the server creates an object, and the client has to free it, or someone else quickly adds a member variable to a class and forgets to clean it up in the destructor. The fact that you even have to do this in the first place is the problem.
> *Absolutely. Java is easy, quick and dirty. You can RAD things with great ease. If development time is an issue, and you don't have the needed libraries in C, then Java is a great language. Just as long as you don't need speed or to get to the nuts-and-bolts of things*
No, my point is that Java is a cleaner language. Packages instead of ugly namespacing. If you really need the speed, you can link C shared libraries directly to your Java code through JNI (Java Native Interface) for that 5-10% of code that really does need to be fast, or bit twidling.

Re:Java is inefficient by **Merk**
"*And Javascript is even worse.*" Javascript really has nothing to do with Java. It's an untyped, interpreted language with a syntax vaguely similar to Java. But there are things you can do with Javascript that you can't do in C. Lambda forms: "new Function("x", "x+6")", runtime modification of objects, etc.
"*I want to pass a variable to a function and have it modify it, oops, no pointers.*" Objects in Java are passed by reference, built-in types aren't, but it's trivial to wrap them in an object if you want a function to modify them.
Re garbage collection: "*I never did understand this gripe. Whenever I put a malloc, I immediately put a free. Whenever I do a new, I immediately put the destroy somewhere.*" Ok, but what about when the memory is allocated by a 3rd party library you're using? What if it is badly documented and doesn't explain which functions allocate memory and which don't? What about exceptions and errors? Sometimes knowing when/where to delete memory is a very complex process, and really, is this something the average programmer should be doing, or something a compiler/runtime/vm should be doing?
Maybe I'm just lazy but I prefer to design a system rather than worry about memory. I've done tight C/C++ code (on the Palm Pilot among other things), and I've done Java. To me, Java is just less frustrating, and places fewer barriers in my way.
C/C++ is really C/C++/preprocessor. When I include a file I don't want

to care if it has already been included, but C/C++ requires that every header file be protected with ifdefs. Why can't the system take care of that for me? What about changing functions? In Java I change it one place, in C/C++ I have to change the header and the source. Why can't the system take care of that for me? I guess I'm just lazy but to me those things are minutia that I don't want to have to bother with. I realize that Java handcuffs me, and I would never choose it for something that had to be highly optimized or really small. At the same time, I'm glad it handcuffs other less skilled developers. If nobody can use pointers, goto statements, global variables, and other messy things, it makes maintaining code so much nicer.
My biggest problem with Java is with the implementation, not the language. If it were designed from the start to be platform-independant code, but code that had to be recompiled for each platform, that would be great. That would fix most of the speed issues, except for the garbage collection.
About beauty: Generally inline assembly, preprocessor junk, goto statements, pointer arithmetic, etc. are ugly code. Sometimes they have their place, but unless they're well documented they're really nasty. Because of this I think it's much easier to write beautiful code in Java.

Java language misconceptions by **yerricde**
>*I want to make a linked list in Java. Ooops, no pointers, sorry.*
As Procrasti mentioned, every variable in the Java language not of primitive type (int, etc.) acts as a pointer. Just because you don't see a * doesn't mean it isn't a pointer.
> *I want to pass a variable to a function and have it modify it, oops, no pointers.*
So pass a reference. If you're passing an object, don't clone() the object before you pass it. If you're passing a primitive, wrap it in an object (i.e. int foo; ... Integer bar = new Integer(foo);).
> *I want to write a program that takes as little memory as possible, or reuse memory, or optimize it to use common options of the processor, oops, no memory management, no assembler.*
Reuse memory by calling System.gc(). Write assembly language either with Jasmin [nyu.edu] (an assembler for JVM bytecode) or JNI (a way to link in unsafe native code).
[...]
> *Damnit, I want a programming language that gives me access to the freeking carry flag! =). I've done math routines a lot, and the code is literally 10x faster when you can optimize it by hand in assembly.*
Then design a language that does such a thing. GCC is free software; you can start from that. And if you don't like the quality of optimizations that GCC does on your code, contribute a better optimizer.

"inefficient" languages by **Tom7**
Right on. I'll gladly take a safe memory-managed language like Java to

write my efficient code, since it means I can spend more time optimizing things which really matter (algorithms).By the way, there exist languages with the same abstraction qualities as java (safety, garbage collection, portability, etc.) which aren't slow in the sense that the original poster meant. Check out O'Caml, for instance. It's got a lot more interesting features than Java, and runs as fast as C.

A long but very enlightening discussion, I think, particularly for those who believed that since programming is a technical matter, it should be possible to agree on what is ‘better’ and settle this sort of discussions once and for all.

53 #2252862, emphasis in the original

II

Annexe

LINE BREAK PRINTING

The exchange presented here is interesting mainly because it shows how questions of code elegance can be dealt with by programmers. But also as a proof of the kind of help (and the speed of it) you may expect to receive from total strangers, people with whom you only share one thing: writing code in a particular prorgamming language (in this case, Perl). (From the site www.perlmonks.org)

breaking a line on printing by **hotshot** on Mar 18, 2002 at 21:41
hotshot has asked for the wisdom of the Perl Monks concerning the following question:
I'm sure someone has answered this already (I couldn't find it in the history), but here goes (a stupid one i'm sure):
How do I use the print command in a way that my code will be more organized, for example:If i want to print a long (more then a row) sentence then my code looks ugly, something like this:

```
print "Once upon a time there was a little programmer that asked to ma
+ny questions.\nToday his question was realy a stupd one.\nand so on an
+d on...";
if ($bla eq 'test') {
   ....;
}
```

and you can see this looks ugly. is there a line seperator or something else to handle these situation of long lines of code (actually not only in print command)?Thanks

Re: breaking a line on printing by **Biker** on Mar 18, 2002 at 21:51
What you're really asking for is a way to define a quoted string on several lines.As far as I'm aware there is no direct way of doing this. (Like in VB or some such.) You could take a different approach:

```
print
"This is a long string\n".
"on more than one\n".
"line\n";
```

This implies that perl will have to do the concatenation, which 'feels wrong'. But the Perl interpreter will optimize this during the compilation phase and create one long string to be printed.

Re: breaking a line on printing by **Tyke** on Mar 18, 2002 at 22:06
Have you looked at the 'heredoc' notation?

```
#!/usr/bin/perl
use strict;
use warnings;
print <<_END_;
Once upon a time there was a little programmer that asked too many que
+stions.
Today his question wasn't realy a stupid one.
And so on and on..
_END_
```

Not a perfect solution with regards to code indentation, but still very useful

Re: breaking a line on printing by **demerphq** on Mar 18, 2002 at 22:12
There are a few variations of Bikers answer
Set $, to be a newline (or use join explicitly as suits your taste and style), also set $\ to a newline.

```
{ # setting $, $\ should be done in a block with local
     local $\="\n";
     {
        local $,="\n";
        print "These",
              "Are",
              "Seperate",
              "lines";
     }
     print join ("\n",
                 "As",
                 "Are",
                 "These");
}

Use a here doc.  (Consult the docs, here docs can be a bit tricky.)

print <<END_OF_TEXT;
These
are
seperate
lines
END_OF_TEXT
```

Yves / DeMerphq

Re: breaking a line on printing by **tachyon** on Mar 18, 2002 at 22:14
You can do any of these:

```
# using a heredoc
print <<TEXT;
Once upon a time there was a little programmer that asked to many ques
+tions.
Today his question was realy a stupd one.
and so on and on...
TEXT

# using literal newlines in the text
print 'Once upon a time there was a little programmer that asked to ma
+ny questions.
Today his question was realy a stupd one.
and so on and on...
';

# using the comma operator
print "Once upon a time there was a little programmer that asked to ma
+ny questions.\n",
      "Today his question was realy a stupd one\n",
      "and so on and on...\n";

# using the concatenation operator
print "Once upon a time there was a little programmer that asked to ma
+ny questions.\n" .
      "Today his question was realy a stupd one\n" .
      "and so on and on...\n";

# using a custom sub. declare prototype so we can use it as
# a bareword. We use ^\n to mark our line wraps and remove
# the leading whitespace on the next line. We use a regex
# to remove these and presto.....
sub wrap;
print wrap "Once upon a time there was a little programmer ^
             that asked to many questions.\nToday his question ^
             was realy a stupd one\nand so on and on...\n";
sub wrap {
    $text = shift;
    $text =~ s/\^\n\s*//g;
  return $text;
}
```

Update
You can even define your own custom print function that does the (un)wrap bit for you:

```
sub printw;
$interpolate = "Interpolate this\n";
$comma = "We can use the comma operator\n";
@ary = qw (this_ is_ an_ array);
printw "Once upon a time there was a little programmer ^
         that asked to many questions.\nToday his question ^
         was realy a stupd one\nand so on and on...\n^
         $interpolate^
         ", $comma, @ary;
sub printw {
    my @text = @_;
    s/\^\n\s*//g, print for @text;
}
```

cheerstachyon

Re: breaking a line on printing by **webadept** on Mar 18, 2002 at 23:59
All of the above are great.. I didn't notice anyone using my favorite

```
print qq` This is text and I want to
print this text on many lines as I can with "quotes and without quotes
+" ..
`;
```

I use that for a lot of HTML stuff and such things. That's not a single quote or apostorphe there.. is the .. heck I don't really know what its called.. its on the same key as the ~ is but I believe you can use just about anything as long as that something is not inside the text you are using. Oh.. and it holds formating.. so watch your line breaks. Try it out, you'll see what I mean.
hope that helps
Glenn H.

Re: Re: breaking a line on printing by **friedo** on Mar 19, 2002 at 01:22
That's a backtick.

Re: Re: Re: breaking a line on printing by **mt2k** on Mar 19, 2002 at 02:25
a backtick?? Oh no! I hate ticks, burn it off, burn it off!! And as for webadept's answer, that is also one of my favorites. I generally use something like

```
#!/usr/bin/perl -w
use strict;
print qq|Content-type: text/html\n
<html>
<head>
<title>Yeah Yeah</title>
</head>
<!-- More HTML stuff here, etc... -->
Oh looky here, a whatever this character is called \|
|;
```

As long as you backslash any |'s in the text you'll be fine (and I don't think your output contains many if any of these...)
If you are outputting strings that contain a lot of strange characters, I'd definitely suggest use heredocs, especially if you are outputting very long HTML documents. For example:

```
print <<'YayIAmDone';
look at this text. lots of non-alphanumeric characters in this print s
+tatement! 8)^404`76`5`65^%!^%^@éA¿º¿ªÑñç_ and we don't even have to lo
+ok for anything to backslash! !$#!#!1~!````~!~!~21&$3(&6(*7_*(+|}|]{]{
+]":':?,,?<
YayIAmDone
exit;
```

Re: breaking a line on printing by **code_drifter** on Mar 19, 2002 at 02:17
I just hit Enter rather than using the "\n". Perl will use the literal keystroke of the enter key.

Re: breaking a line on printing by **abaxaba** on Mar 19, 2002 at 05:12
There have been some good suggestions here, but no one has mentioned my particular favorite, which I use especially with html output. I push everything onto an array, then just dump the array. This seems to work for me for a couple of reasons: Format whitespace, easier readability.
Pass array ref around to different subs, to build up output string in a modular format, without worrying about return values:

```
#!perl
main();
sub main
{
    my @output;
    push (@output, "Here is something to print out\n",
                   "Here is yet another line\n");
    extra(\@output);
    print @output;
}
sub extra
{
    my $resRef = shift;
    push (@$resRef, "Here is an extra line\n");
}
exit 0;
```

Re: breaking a line on printing by **ellem** on Mar 19, 2002 at 05:28
You can do all of the wonderful things everyone has already suggested BUT if I read you question correctly you are concerned that your code is ugly... (strange for a Perl programmer.. but OK =))

```
print "I want this on one line
I want this on another line
and finally put this here on yeat another line.\n"
```

It's practically WYSIWYG. Just hit enter at the end of what you're doing.

Re: Re: breaking a line on printing by **MAXOMENOS** on Mar 19, 2002 at 11:29
I'm surprised more people haven't suggested this approach. I see people do this all the time, without ill effects. Has anyone seen this *not* work?

III

Annexe

THE MOST BEAUTIFUL PIECE OF CODE

The following thread is one of the longets in the discussion, it was started by *mduell*, who suggested a very simple program. But despite its straightforwardness, the program triggered all kinds of reactions. The last messages in the thread are written by assembler fans, i.e. people who prefer to write code without the help of (high level) programming languages, but it would seem one of them was only pretending…

The most beautiful piece of code... by **mduell** (#483118)

```
#include <stdio.h>
int main( void )

{
printf( "Hello world!" );
}
```

Mark Duell

Re:The most beautiful piece of code... by **Anonymous Coward** (#482974)
Tsk tsk. You forgot the "return(0);". And you didn't indent. AND you didn't check the return value of printf() (forgiveable, everyone forgets that). Is this what beauty's about? That particular snippet is ugly enough to warrent a warning in most compilers, but slashdot seems to think its worth a +1...

Re:The most beautiful piece of code... by **Girf** (#483160)
•Actually, I find that piece ugly. If I were wanting a beautiful 'Hello World' I would:
•Lose the '!'. Exclaimation marks are lame; there isn't really much to get excited about.
•Tabs, need I say more?
•Use a '\n'. I really hate when a program makes my console look like Hello world![james@jimbob ~]. Any code/coder that doesn't use line breaks should be whipped and beaten then sent back to Visual Basic where they belong.

Re:The most beautiful piece of code... by **thelaw** (#483159)
> sent back to Visual Basic where they belong.
or tcl.
jon

Re:The most beautiful piece of code... by **nycsubway** (#483127)
its beautiful, but aren't you missing the return value? since main is an int...

Re:The most beautiful piece of code... by **RoninM** (#483167)
How about:

```
#include <stdio.h>
int main(void)
{
     (void)printf("Hello, world!\n");
}
```

?

Re:The most beautiful piece of code... by **maw** (#483068)
You shouldn't use tabs in code. (The exception that makes the rule is that Makefiles require tabs.)
It's better, in a cross-platform portability way, to use individual spaces. If somebody is using an editor which can't automatically change the number of spaces, too bad for him.
Obviously, indentation is important.

Re:The most beautiful piece of code... by **Tassach** (#483209)
You shouldn't use tabs in code. (The exception that makes the rule is that Makefiles require tabs.)
Ah, yet another holy war, right up there with vi vs emacs. Personally, I hate working on code indented with spaces. I'll admit that it's annoying to edit tabbed code on a broken editor; but the way to solve that problem is to fix the editor, not the code.

Re:The most beautiful piece of code... by **maw** (#483069)
Ah, yet another holy war, right up there with vi vs emacs.
No, I maintain that it is not a holy war: holy wars always concern personal preference; the tabs vs spaces debate is one of technical interoperability.

...but the way to solve that problem is to fix the editor, not the code.
I disagree, and rather than repeat the arguments myself, I point you here.

Re:The most beautiful piece of code... by **joto** (#483199)
No, it's not beautyful.
1. It is not even legal ISO (or ANSI) C, because main should return a value!
2. It is stupid to cast the return value from printf(). It introduces more visual clutter, and serves no purpose.
3. I think you could afford a line of whitespace between the preprocessor directive and the main function.
4. It does nothing useful.

Re:The most beautiful piece of code... by **RoninM** (#483171)
2. *It is stupid to cast the return value from printf().*
That's not true. printf() returns an int. Casting it to void is more correct than silently throwing away the return value.
3. *I think you could afford a line of whitespace...*
Good for you. That's a matter of style. The program is not more or less beautiful or elegant because of it.
But, point taken on 1. My oversight.

Re:The most beautiful piece of code... by **joto** (#483203)
That's not true. printf() returns an int. Casting it to void is more correct than silently throwing away the return value.
No. It is not "more" correct. In fact, both options are legal ISO C, and therefore equally "correct". It is, however, a stylistic issue.
And when it comes to style, opinions sometimes differ. I agree that there might theoretically exist situations were a void-cast could theoretically improve some readers understanding of a program, but I have yet to see that in practice. Anyone knows that printf() is called mainly for a side-effect. And side-effecting functions should not be a foreign concept to C programmers, as C is not exactly what I would call a pure functional language.
Anyway, I think any C-programmer on the planet knows that printf() is called mainly for a side-effect. You do not need to tell them that with a void-cast, as little as you need to tell them that with a comment. Do you really think there is even a single programmer on the planet that think it is easier to understand your programs because you put in lots of redundant unnesseceary casts?
Good for you. That's a matter of style. The program is not more or less beautiful or elegant because of it.
Beautyful? Yes, Elegant? No

Re:The most beautiful piece of code... by **agentZ** (#483278)
4.*It does nothing useful.*
Nonsense! It tells you that the compiler works.

```
int main() {
   return (printf("hello world\n"));
}
```

Re:The most beautiful piece of code... by **naasking** (#483145)
No, it's not beautyful. It is not even legal ISO (or ANSI) C, because main should return a value!
bzzzt! wrong! Checking out my trusty ANSI C book, the simple program "Hello World!" (or any other program for that matter) does NOT require you to return a value at the end of main to indicate success(though some older compilers require it). If you get to the end of the main block, that is assumed to be grounds for correct correct

program termination(so the compiler will helpullfy insert the return statement for you).

Re:The most beautiful piece of code... by **joto** (#483202)
bzzzt! wrong! Checking out my trusty ANSI C book
Maybe you need to check out another C book then. Let my guess, you are using The Annotated ANSI C Standard, annotated by Herbert Schildt? This is probably the worst book ever written on the ANSI C standard. Or are you just using some other half-good book on C? I doubt you are actually using the ANSI standard, because in that case you have proven that you do not know how to read.
If you get to the end of the main block, that is assumed to be grounds for correct correct program termination(so the compiler will helpullfy insert the return statement for you).
No, that is not true ISO C. I think it might be true of C++, but then again, that's a completely different language. Also, the fact that some compilers will allow it, is not very interesting either, since compilers are allowed to do what ever they want when it comes to undefined behaviour, which is what this is.

Re:The most beautiful piece of code... by **naasking** (#483147)
No, it was a decent ANSI C book as far as I know. "The C programming Language" by Brian W. Kernighan and Dennis M. Ritchie. Perhaps you've heard of it? ;-)

Re:The most beautiful piece of code... by **mduell** (#483119)
Ok, in my rush to post, I forgot to #include <stdlib.h> and add return EXIT_SUCCESS to the end.
Mark Duell

Re:The most beautiful piece of code... by **orangesquid** (#483128)

```
if(!printf("Hi there.\n")) {
  if(puts("printf() didn't print anything!")==EOF) {
    perror("puts() didn't print \"printf() didn't print anything!\" because
your
screen ran out of space(?)...!");
      /* doesn't return anything, so nothing to check... finally! */
   }
}
```

Re:The most beautiful piece of code... by **commanderfoxtrot** (#483176)
The most elegant simple language was BBC BASIC, written by Acorn and used for the official BBC computers. Very simple, yet could do some amazing things, especially given it was written at the end of the 1970s. Here we go:

```
PRINT"Hello World."
```

That's it!

Re:The most beautiful piece of code... by **Abreu** (#483239)
Even easier:

```
#!/usr/bin/python
print "Hello World"
```

Remember to chant this over and over while you code:
Beautiful is better than ugly... Explicit is better than implicit... Simple is better than complex...Complex is better than complicated... Flat is better than nested... Sparse is better than dense... Readability counts... Special cases aren't special enough to break the rules... Although practicality beats purity... Errors should never pass silently... Unless explicitly silenced... In the face of ambiguity, refuse the temptation to guess... There should be one-- and preferably only one --obvious way to do it... Although that way may not be obvious at first unless you're Dutch... Now is better than never... Although never is often better than *right* now... If the implementation is hard to explain, it's a bad idea... If the implementation is easy to explain, it may be a good idea...

Re:The most beautiful piece of code... by **domc** (#483031)

```
#!/usr/bin/perl -w
print "Hello World\n";
```

Re:The most beautiful piece of code... by **Bungie** (#483253)
Two lines:

```
PRINT "Hello World"
END
```

Re:The most beautiful piece of code... by **garett_spencley** (#483256)
How about:

```
msg:
     .string         "Hello, world!\n"
     .globl main
main:
     pushl %ebp
     movl %esp,%ebp
     subl $8,%esp
     addl $-4,%esp
     pushl $14
     pushl $msg
     pushl $1
     call write
     addl $16,%esp
     xorl %eax,%eax
     jmp return
     return:
     movl %ebp,%esp
     popl %ebp
     ret
```

????
I know it's not indented... I tried... Darn html.. Oh well...
--
Garett

Re:The most beautiful piece of code... by **StarDrifter** (#483221)

```
subl $8,%esp
addl $-4,%esp
```

You cheated! This code was produced by gcc with slight modifications to remove the obvious compiler generated names and directives. No beautiful hand-coded asm program can have those two lines next to each other in the source.

Re:The most beautiful piece of code... by **lpontiac** (#483241)
Try this:

```
msg:
        .string "Hello, world!\n"
        .global main
        .type   main,@function
main:
        pushl    $14
        pushl    $msg
        pushl    $1
        call     write
        addl     $12, %esp
        movl     $0, %eax
        ret
```

You see, there's no *need* to save and restore ebp if you're nice to the stack. Additionally, you could use a linux system call (via int 0x80) instead of calling libc's write function, but the use of main kind of ties us into a C infrastructure anyways and that would just break compatability on other x86 Unices.

Re:The most beautiful piece of code... by **QZS4** (#483019)
Nah, try this:

```
.data
    msg: .asciiz "Hello, world!\n"
.text
    .globl main
    .ent main
main:
    subu sp,sp,4
    sw ra,0(sp)
    la a0,msg
    jal printf
    lw ra,0(sp)
    addiu sp,sp,4
    jr ra
    .end main
```

Or, why not this:

```
.section ".rodata"
msg: .asciz "Hello, world!\n"
.section ".text"
    .global main
    .type main,#function
main:
    save %sp,-112,%sp
    sethi %hi(msg),%o0
    or %o0,%lo(msg),%o0
    call printf, 0
    nop
    ret
    restore
```

But my Sparc assembly is a little rusty, so the last one might not be entirely correct.

Re:The most beautiful piece of code... by **neotek(maas)** (#483292)
To hell with compatibility.

```
.globl _start
     .type _start,@function
_start:
     jmp 0xd
     pushl $0x6f6c6c65
     andb %dh,0x6f(%edi)
     jb 0x6c
     orb %fs:(%eax),%al
     movl $0xc, %edx
     movl $_start, %ecx
     addl $2, %ecx
     movl $0x1, %ebx
     movl $0x4, %eax
     int $0x80
     xorl %ebx,%ebx
     movl $0x1, %eax
     int $0x80
```

Re:The most beautiful piece of code... by **Deflatamouse!** (#483195)
looks like a piece of $h!t to me

IV
Annexe

NOT THIS STUPID
PROGRAMMING IS ART BS AGAIN

In this thread, participants argued about whether it is correct to call programming an art. In the end, nothing was clearly settled, even if it would seem that all agreed that programming is, at least, a craft. This debate has also been brought up in more academic texts (Bolter 1993; Dahlbom and Mathiassen 1993).

Not this stupid 'programming is art' BS again! by **Flabdabb Hubbard** (#2252879)
to recognize the artistry involved in writing software
What pretentious bullshit. Software is NOT art. It can be closely compared to bricklaying, or cabinet making, it is a CRAFT.
Try expressing an emotion in C++. It cannot be done. Please think before repeating these banal opinions that software is art. It just isn't. Deal with it, and if you want to be an artist, learn to paint.

Re:Not this stupid 'programming is art' BS again! by **FortKnox** (#2252918)
Now now. Art can be seen in any type of medium, even coding.
You may not understand it, but only the artist needs to.
I can see art in some obfuscated code I've seen.
Who are you to tell me what is art and what isn't?

Re:Not this stupid 'programming is art' BS again! by **servo8** (#2252939)
I think a problem here is getting to a common definition of art. If a master craftsman pours his soul into a work, how is that not art? Just because the emotions a work may convey cannot be easily categorized and labelled does not mean they are not valid feelings. There are many pieces of "craftsmanship" out there that evoke such feelings. I have felt them myself. Would you deny me that?

Re:Not this stupid 'programming is art' BS again! by **DCheesi** (#2253085)
Crafts are made *primarily* for practical use, often with aesthetics as a major secondary consideration. Art is made primarily for art's sake, to stimulate thought & emotion. The reason that the lines seem to blur so often these days is that we have so many choices in products that we often select them solely on the basis of aesthetic value, even though the objects are ultimately meant to serve a practical purpose.
ObOnTopic: Software is indeed a craft, and can be approached from many points of view. But the only code that is truly 'art' is that which is written primarily for the appreciation of other programmers. Real-world software doesn't, and shouldn't, fall into this category.

Re:Not this stupid 'programming is art' BS again! by **Glock27** (#2253020)
What pretentious bullshit. Software is NOT art. It can be closely compared to bricklaying, or cabinet making, it is a CRAFT.
Very perceptive...coding software is like crafting a cabinet. However, designing a cabinet is art...and so is designing software.
Try expressing an emotion in C++. It cannot be done.

```
jesus->loves(you);  //  Sarcasm, for the humor impaired
```

Regardless, art doesn't just express emotion, it inspires emotion. And trust me, I've had (mostly other people's) C++ code inspire some pretty horrific emotions. ;-)
Good design and coding, on the other hand, can truly be things of beauty, regardless of language.
Please think before repeating these banal opinions that software is art. It just isn't. Deal with it, and if you want to be an artist, learn to paint.
Spoken like someone who just doesn't really comprehend software design, or why one design might be more elegant than another. I suppose you don't think mathematics is beautiful either...
186,282 mi/s...not just a good idea, its the law!

Re:Not this stupid 'programming is art' BS again! by **Peyna** (#2254373)
Beauty and Art are two different things.
(i.e.) I think a Porsche would be beautiful in my garage, but that doesn't mean a Porsche is a work of art.
I suppose if you were going to compare coding with art (literature), coding would be non-fiction. Some of it is horribly written, but it's got all the facts right. Some of it is written wrong and doesn't make any sense. While the truly good works of non-fiction, even though they are only telling you facts, are beautiful, fill your mind and heart with thoughts and feelings, and are 100% accurate at the same time.
In that sense, yes it is art, and like with a non-fiction book, you can copyright the finished product, but you can't copyright the facts that make it up.

Re:Not this stupid painting is art' BS again! by **Anonymous Coward** (#2254886)
Just because you can paint, doesnt mean you're an artist, I've known lots of builders, painters and decorators, and none of them were artists!
When are they gonna grow up, and go out and buy a computer and write some truely beautiful
code????
eh? eh? eh?
come on then!

Re:Not this stupid 'programming is art' BS again! by **jeremyp** (#2255076)
Programming is not an art, cabinet designing is not an art, architecture is not an art. I'm defining the term "artist" quite narrowly as somebody who makes objects who's primary purpose is to inspire some sort of emotional response from people. e.g. a Henry Moore sculpture has no other purpose.
Software, cabinets and buildings have a different primary purpose and subjugating that purpose to the goal of making it look nice often results in less than optimum performance in the primary purpose. e.g. an architect may decide to make his building entirely of glass for aesthetic reasons, but that might make it too hot in sunny weather. Another example would be the original design for Quicktime 4 which looked great with its brushed metal graphics, but was terrible to use (review here http://www.iarchitect.com/qtime.htm).

Re:Not this stupid 'programming is art' BS again! by **chris_mahan** (#2255424)
[...] Now, I write code. I want to make the user feel a bond with a freaking motherboard. If I succeed in making a grown man or woman "enjoy the interaction" with a piece of plastic/metal/goo, and I have done that on purpose, is that not art?
I contend that in the same way the common man does not recognize Beethoven's 5th symphony by looking at the sheet music, likewise the common man does not recognize great, beautiful, engaging, pleasing software by looking at source code.
There are millions of programmers in the world who consider source code to be art, to be speech. Who are non-programmers to say that it isn't?

Re:Not this stupid 'programming is art' BS again! by **Anonymous Coward** (#2255136)
Is coding by itself even a craft? It's really nothing more than translating an algorithm from one form to another. Under copyright law, I can make a case that *coding* is nothing more than a work-for-hire.
I find the arguments against coding standards really lame. "It hurts my creativity".
Buddy, if your creativity depends on the placement of braces and how many spaces you use to indent, you're a real lamer.
Designing, on the other hand, *is* art.

Re:Not this stupid 'programming is art' BS again! by **Anonymous Coward** (#2257073)
Actually, programming (like engineering, metallurgy and materials sciences) is both an art and a science, when using classical definitions. The "science" part comes in because if you repeat a set of actions, you get

the same results. The "art" part comes in when you select *which* set of actions will produce the results you want, while balancing various limited resources against each other. In fact, it can be a "black art" when you don't know why a set of actions produce the results it does. It is also a craft, in the classical sense. Just about anybody can learn the basic rules (I can build a chair), but not many people can get really good at it (I can't make an elaborate chair with inlays and carvings), and only a few people can make works of greatness (how many people can make a throne?).
Finally, don't forget that Univerities produce "Artists" every time a person graduates with a BA or an MA (the 'A' stands for "of Arts").

You're an idiot. by **Anonymous Coward** (#2253105)
Well, you've obviously never designed or written software.
I would agree that bricklaying is not art, but how about architecture? The bricklaying of programming is the actual typing.
I've expressed emotions in C++. Fuck, I've written outright poetry in perl.
To play devil's advocate:
What pretentious bullshit. Painting is NOT art. It can be closely compared to putting dots of color on a piece of paper, it is a CRAFT.
Try expressing an emotion with dots of color on paper. It cannot be done. Please think before repeating these banal opinions that painting is art. It just isn't. Deal with it, and if you want to be an artist, learn to hack perl.

Re:Not this stupid 'programming is art' BS again! by **Anonymous Coward** (#2254438)
I can make the reader cry with one line of C:

```
void main(void)
```

I can make the reader angry with one line of most any language:

```
goto HELL
```

With code, one can make jokes, poems, mysteries, ironies, or metaphors--not that these make good code. A creative person can use any medium.

Re:Not this stupid 'programming is art' BS again! by **big_hairy_mama** (#2254476)
IANACM (I'm not a cabinet maker), but I've seen some very beautiful cabinets in my time... I would therefore say that it is definitely an art. And I've also seen some very beautiful code in my time; in fact some code I would readily compare to the Mona Lisa, while other code I would compare to a pre-schooler's fingerpainting.

On a mundane level, coding is a craft. But if you treat it like an art (and know what you are doing), then it is exactly that.

Re:I Can see the point... by **matek** (#2253008)
And so we come to the definition of ART.
Some would say that art is when someone expresses their feelings in some form.
Other would say that art is when someone makes something that looks/sounds/feels very nice..
Source code can be art-like. Sometimes, when you see some beatyfull software design, the tears start to fall.. IMHO code can be a modern form of art - just like bodypainting is an art...

Re:nice, but welcome back to the real world by **jbum** (#2252929)
> I need it to work, not look good.
Hear hear. Engineers with an over-developed aesthetic sense are writing their code for other engineers, not the end-user. Too many times in my professional life have I seen inordinate amounts of time wasted on issues which are invisible to the end-user, because some overly- aesthetically minded engineer couldn't sleep at night.
It's a craft, not an art; and if you can't sleep at night, try getting laid.

Bibliography

ASPLUND, J. 1970 *Om undran inför samhället*, Lund: Argos.

— 1987 *Det sociala livets elementära former*: Bokförlaget Korpen.

ASSOCIATION FOR COMPUTING MACHINERY 1997 'Papers presented at the seventh Workshop on Empirical Studies of Programmers', New York, NY: Association for Computing Machinery.

BATAILLE, G. 1988 *The accursed share : an essay on general economy*, New York: Zone Books.

BENTLEY, J. L. 1986 *Programming pearls*, Reading, Mass: Addison-wesley.

BIJKER, W. E. AND LAW, J. (eds) 1992 *Shaping Technology / Building Society: Studies in sociotechnical change*: Massachusetts Institute of Technology.

BOLTER, J. D. 1993 *Turing's Man - Western culture in the computer age*, Harmondsworth: Penguin Books.

BROOKS, F. P., JR. 1995 *The Mythical Man-Month: Essays on software engineering*, Boston: Addison Wesley Longman, Inc.

BRYMAN, A. ANDNILSSON, B. 2002 *Samhällsvetenskapliga metoder*, 1. uppl. Edition, Malmö: Liber ekonomi.

CARRASCO, D. 1999 *City of Sacrifice: the aztec empire and the role of violence in civilization*, Boston: Beacon Press.

CERRUZI, P. E. 2000 *A History of Modern Computing*, Cambridge, Massachusetts: MIT Press.

CONNELL, C. H. 2001 'Software Stinks!'
www.chc-3.com/pub/beautifulsoftware.htm.

COOK, C. R., SCHOLTZ, J. C. AND SPOHRER, J. C. 1993 *Empirical studies of programmers : fifth workshop : papers presented at the Fifth Workshop on Empirical Studies of Programmers, December 3-5, 1993, Palo Alto, CA*, Norwood, N.J.: Ablex Pub. Corp.

COOPER, A. 1999 *The Inmates are Running the Asylum*, Indianapolis: SAMS.

DAHLBOM, B. AND MATHIASSEN, L. 1993 *Computers in Context: The philosophy and practice of systems design*, Cambridge: NCC Blackwell.

DIJKSTRA, E. W. 1968 'Go To Statement Considered Harmful', *Communications of the ACM* 11(3): 147-148.

— 1972 'The humble programmer', *Communications of the ACM* 15(10): 859-866.

ELIAS, N. 2000 *The Civilizing Process*, Oxford: Blackwell Publishing.

ELLUL, J. 1964 *The Technological Society*, New York: Vintage Books.

FEENBERG, A. 1999 *Questioning technology*, London ; New York: Routledge.

FEYERABEND, P. K. 1978 *Against Method*, London: Verso Editions.

— 1987 *Farewell to reason*, London ; New York: Verso.

FLORMAN, S. C. 1994 *The Existential Pleasures of Engineering*, New York: San Martin's Griffin.

GABRIEL, R. P. 1996 *Patterns of Software*, Oxford: Oxford University Press.

GABRIEL, R. P. AND GOLDMAN, R. *Mob Software: The erotic life of software*, OOPSLA 2000, Keynote.

GANCARZ, M. 1995 *The UNIX philosophy*, Boston: Digital Press.

GEERTZ, C. 1973 *The interpretation of cultures; selected essays*, New York,: Basic Books.

— 1993 *Local Knowledge*, London: Fontana Press.

GELERNTER, D. H. 1998 *The aesthetics of computing*, London: Weidenfeld & Nicholson.

GOFFMAN, E. 1990 *The Presentation of Self in Everyday Life*, London: Penguin Books.

GUILLET DE MONTHOUX, P. 1981 *Doktor Kant och den oekonomiska rationaliseringen : om det normativas betydelse för företagens, industrins och teknologins ekonomi*, Göteborg: Korpen.

— 1998 *Konsföretaget*, Göteborg: Bokförlaget Korpen.

GUSTAFSSON, C. 1994 *Produktion av allvar : om det ekonomiska förnuftets metafysik*, Stockholm: Nerenius & Santérus.

— 2000 'Känslan av Zap', *Dialoger, Stockholm.*

HEGARTY, P. 2000 *Georges Bataille*, London ; Thousand Oaks, Calif.: Sage.

HEIDEGGER, M. AND KRELL, D. F. 1993 *Basic writings : from Being and time (1927) to The task of thinking (1964)*, Rev. and expanded Edition, San Francisco, Calif.: HarperSanFrancisco.

HIGGS, E. S., LIGHT, A. AND STRONG, D. 2000 *Technology and the good life?*, Chicago: University of Chicago Press.

HIMANEN, P. 2001 *The hacker ethic, and the spirit of the information age*, 1st Edition, New York: Random House.

HOARE, C. A. R. 1981 'The emperor's old clothes', *Communications of the ACM* 24(2): 75--83.

HOFSTADTER, A. AND KUHNS, R. 1976 *The Philosophies of Art and Beauty*, Chicago: The University of Chicago Press.

HUBERT, H. AND MAUSS, M. 1964 *Sacrifice: its nature and functions*, Chicago: University of Chicago Press.

HUIZINGA, J. 1955 *Homo Ludens: A study of the play element in culture*, Boston: Beacon Press.

HUNT, A. AND THOMAS, D. 2000 *The pragmatic programmer : from journeyman to master*, Reading, Mass.: Addison-Wesley.

HYDE, L. 1983 *The gift : imagination and the erotic life of property*, 1st Vintage Books Edition, New York: Vintage Books.

JEFFRIES, R. E. 2001 'What is Extreme Programming?'

KANT, I. 1957 *The critique of judgement*, Oxford: Clarendon Pr.

KERNIGHAN, B. W. AND PIKE, R. 1984 *The Unix Programming Environment*: Prentice-Hall Software Series.

KIDDER, T. 1981 *The soul of a new machine*, Boston: Little Brown.

KNUTH, D. E. 1974a 'Computer programming as an art', *Communications of the ACM* 17(12): 667-673.

— 1974b 'Structured Programming with go to Statements', *Computing Surveys* 6(4).

— 1983 'Literate Programming', *The Computer Journal (submitted to)*.

— 1992 *Literate programming*, Stanford, CA: Center for the Study of Language and Information.

— 1997 *The art of computer programming*, 3rd Edition, Reading, Mass.: Addison-Wesley.

— 1998 'Adventure',
http:/www.literateprogramming.com/adventure.pdf.

KOHANSKI, D. 2000 *Moths in the Machine: the power and perils of programming*, New York: St. Martin's Griffin.

KUNDA, G. 1991 *Engineering culture : control and commitment in a high-tech corporation*, Philadelphia: Temple University Press.

LANGER, S. K. 1957 *Philosophy in a New Key*, Cambridge: Harvard University Press.

LATOUR, B. 1987 *Science in action : how to follow scientists and engineers through society*, Cambridge, Mass: Harvard University Press.

— 1996 *Aramis, or, The love of technology*, Cambridge, Mass.: Harvard University Press.

LEVY, S. 1984 *Hackers : heroes of the computer revolution*, 1st Edition, Garden City, N.Y.: Anchor Press/ Doubleday.

LINDBLOM, C. 1959 'The Science of Muddling Through', *Public Administration Review* 19: 79 -- 88.

LOHR, S. 2002 *GOTO - Software Superheroes, from Fortran to the Internet Age*, London: Profile Books Ltd.

MACINTYRE, A. 1985 *After Virtue: A study in moral theory*, London: Duckworth.

MAUSS, M. 1967 *The gift : forms and functions of exchange in archaic societies*, New York: Norton.
MCCANN 2001 'Style Principle'.
MCCONNELL, S. 1996 *Rapid Development: Taming Wild Software Schedules*, Redmond: Microsoft PRess.
MILLER, D. 1998 *A Theory of Shopping*, Cambridge: Polity Press.
MITCHAM, C. 1989 *¿Qué es la filosofía de la tecnología?*, Barcelona: Editorial Anthropos.
MOORE, J. T. S. 2001 'Revolution OS': Wonderview Productions, LLC.
MUMFORD, L. 1952 *Art and technics*, New York,: Columbia University Press.
OMAN, P. W. AND COOK, C. R. 1990 'Typographic Style is More than Cosmetic', *Communications of the ACM* 33(5): 506 - 520.
ORTEGA Y GASSET, J. 1996 *Meditación de la técnica y otros ensayos sobre ciencia y filosofia*, Madrid: Revista de Occidente en Alianza Editorial.
PACCAGNELLA, L. 1997 'Getting the Seats of your Pants Dirty: Strategies for ethnographic research on virtual communities', *Journal of Computer Mediated Communication* 3(1).
PARGMAN, D. 2000 *Code begets community : on social and technical aspects of managing a virtual community*, 1. Edition, Linköping: Tema Univ.
PETROSKI, H. 1992 *To Engineer is Human*, First Vintage Books Edition Edition, New York: Vintage Books - Random House Inc.
PIKE, R. 1989 'Notes on Programming in C', Vol. 2002.
PLOG, F. AND BATES, D. G. 1976 *Cultural anthropology*, New York: Alfred A. Knopf, Inc.
RAYMOND, E. S. 1998 'The Cathedral and the Bazaar': Firstmonday.
— 2000 'Homesteading the Noosphere', Vol. 2003: Firstmonday.

— (ed) 2003 *The Jargon File, version 4.4.4*: http://catb.org/esr/jargon/html/g001.html.

REHN, A. 2001 *Electronic Potlatch : a study of new technologies and primitive economic behavior*, Stockholm: KTH.

RICE, P. 1996 *An Engineer Imagines*, London: ellipsis london limitied.

SAHLINS, M. 1972 *Stone Age Economics*, New York: Aldine de Gruyter.

— 2000 *Culture in Practice - Selected Essays*, New York: Zone Books.

SEAMAN, C. B. AND BASILI, V. R. 1998 'Communication and Organization: An Empirical Study of Discussion in Inspection Meetings', *IEEE Transactions on Software Engineering* 24(6): 559-572.

SHARP, H., ROBINSON, H. AND WOODMAN, M. 2000 'Software Engineering: community and culture', *IEEE Software*(January/February 2000).

SIBLEY, F. 2001 *Approach to Aesthetics, Collected papers on Philosophical Aesthetics*, Oxford: Clarendon Press.

SIMON, H. A. 1997 *Administrative Behaviour*, New York: The Free Press.

SUDWEEKS, F. AND RAFAELI, S. 1996 'How do you get a hundred strangers to agree? Computer-mediated communication and collaboration', in T. M. Harrison and T. D. Stephen (eds) *Computer Networking and Scholarship in the 21st Century University*, New York: SUNY Press.

TAYLOR, F. W. 1967 *The principles of scientific management*, New York,: Norton.

TAYLOR, P. A. 1999 *Hackers*, London: Routledge.

TORVALDS, L. AND DIAMOND, D. 2001 *Just for fun : the story of an accidental revolutionary*, 1st Edition, New York, NY: HarperBusiness.

TOWNSEND, D. 1997 *An Introduction to Aesthetics*, Oxford: Blackwell Publishers.

TURKLE, S. 1997 *Life on the Screen: Identity in the*

Age of the Internet, New York: Simon & Schuster.

ULLMAN, E. 1995 'Out of Time: Reflections on the Programming Life', in J. Brook and I. Boal (eds) *Resisting the Virtual Life: The Culture and Politics of Information*, San Francisco: City Lights Books.

— 1997 *Close to the machine : technophilia and its discontents*, San Francisco: City Lights Books.

— 1998 'The dumbing-down of programming', *Salon.com*.

VALVERDE, J. M. 1998 *Breve historia y antología de la estética*, Barcelona: Editorial Ariel, S.A.

VAN MAANEN, J. 1979 'The Fact of Fiction in Organizational Ethnography', *Administrative Science Quarterly* 24(4): 539 - 550.

VEBLEN, T. 1990 *The Instinct of Workmanship - and the State of Industrial Arts*, New Brunswick, New Jersey: Transaction Publishers.

VIGOTSKIJ, L. S. 1995 *Fantasi och kreativitet i barndomen*, Göteborg: Bokförlaget Daidalos AB.

WEINBERG, G. M. 1971 *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold Company.

WINOGRAD, T. 1996 *Bringing design to software*, New York, N.Y.: ACM Press ; Addison- Wesley.

WITTGENSTEIN, L. 1969 *Preliminary studies for the 'Philosophical investigations', generally known as the Blue and Brown books*, 2nd Edition, Oxford,: Blackwell.

WITTGENSTEIN, L. AND ANSCOMBE, G. E. M. 1997 *Philosophical investigations*, 2nd Edition, Oxford ; Malden, Mass: Blackwell.

WITTGENSTEIN, L. AND BARRETT, C. 1966 *Lectures & conversations on aesthetics, psychology, and religious belief*, Oxford: B. Blackwell.

WRIGHT, G. H. V. 1972 *The Varieties of Goodness*, London: Routledge & Kegan Paul.

— 1994 *Myten om framsteget : tankar 1987-1992 : med*

en intellektuell självbiografi, Ny utg. Edition, Stockholm: Bonnier.
— 2000 *Vetenskapen och förnuftet : ett försök till orientering*, Ny utg. Edition, Stockholm: Bonnier.
Yourdon, E. 1997 *Death March*, Upper Saddle River, New Jersey: Prentice Hall PTR.

"FIELDS OF FLOW"

Art & Business
Aesthetics, Technology & Management

The Fields of Flow research program is a unique joint scholarly undertaking by three academic institutions. The program is being carried out in close cooperation with prominent institutions and organizations in the arts. The 'fields' in focus are art and business. The notion of 'flow' presented in the title of the program draws attention to movement across disciplines and changes, thereby referring to different points of contact and interaction between the fields of art and business.

Art and business are basic building blocks in any social- or social welfare deve-lopment. The two elements presuppose, constitute, and help bring about change in one another. The understanding of art requires an understanding of management – and vice versa. This rationality of simultaneity and coexistence has for some time been controversial – the logic of distinctiveness have instead characterized most of the actions. In recent years, however, there has occurred a reuniting of these fields. We see such cross-fertilization – flows of experience and knowledge – as an essential area of study in gaining new insight into many changes going on in society.

Fields of Flow is being carried out by a core group of about fifteen researchers at three universities, under the leadership of Professor Sven-Erik Sjöstrand (Stockholm School of Economics) together with Professor Pierre Guillet de Monthoux (Stock-holm University) and Professor Claes Gustafsson (Royal Institute of Technology). In addition to the core group, the program involves many others – researchers as well as practitioners – foremost from the field of art. Extensive collaboration has been established with national and international research teams, businesses and art institutions. An Advisory Board comprising about fifteen prominent leaders in the fields of business and the arts is also associated to the program.

The Fields of Flow program builds on three main themes: (1) Artistic and Cultural Production, (2) The Aesthetic Dimensions of Management, Technology and Organization, and (3) Art Meets Business. Business Meets Art. The program includes close to twenty 'semi-autonomous' research projects. They are all described at the homepages of the three core institutions, i.e.) for Stockholm School of Econo-mics www.hhs.se (press Research & Publications button and then the text 'Management and Organization', for Stockholm University www.su.se (look for Business administration and then 'Research'), and for the Royal Institute of Techno-logy www.kth.se (press 'Research').

Erik Piñeiro
Royal Institute of Technology
Department of Industrial Management
Lindstedtsvägen 30, 100 44 Stockholm
Tel. 08 790 79 85, fax 08 790 76 17
www.indek.kth.se

Grafisk form: Gabor Palotai Design
Tryck: Kristianstads Boktyckeri AB, Kristianstad

Arvinius Förlag
Box 6040, 102 31 Stockholm
Tel. 08 32 00 15, fax 08 32 00 95
www.arvinius.se

ISSN 1100-7982
ISBN 91-7283-614-8
TRITA-IEO-R 2003,15
ISRN/KTH/IEO/R-03/12--SE