Understanding and Power in Software Development Practices

Viktor Binzberger

PhD Thesis

Supervisor: Dr. Tihamér Margitay, C. Sc.

Budapest University of Technology and Economics
Faculty of Economic and Social Sciences
2007

Contents

1. Preface

1.1. Short abstract

In the thesis, I propose a novel way to look upon software. Revitalizing the perspective of the hermeneutic tradition of philosophy, I argue that we should look upon software as a region of the phenomenal world of our contemporary life: a shared field of meanings, experiences, interactions, and skillful practices. This perspective draws attention to questions such as: What idealizations of human interaction do we build into our systems? How do these ideals transform our interactions and experiences within the world of software?

To address these issues, I introduce the concepts *hermeneutic practice*, *power field*, and *user appropriation* in the contexts of use and development of info-communication technologies. Instead of giving abstract definitions, I make these concepts meaningful through showing their relevance in detailed examples and case studies.

1.2. Összefoglaló

Megértés és hatalom a szoftverfejlesztés gyakorlatában

Milyen szerepet tölt be a szoftver a megértésben? Milyen rejtett előfeltevéseket, hatalmi viszonyokat kódol a programkód? Hogyan formálják át hétköznapjaink tevékenységformáit a szoftverek, és ez milyen következményekkel járhat? Doktori értekezésemben ezekre a fontos filozófiai kérdésekre keresem a választ.

Megközelítésem kortárs- és történeti esettanulmányokra, valamint különféle filozófiai és szociológiai elméletekre támaszkodik. A doktori értekezésben bevezetem a „hermeneutikai tevékenység", a „hatalmi mező" és a „felhasználói appropriáció" fogalmait, és esettanulmányokon keresztül mutatom be az infokommunikációs technológiák működésének az értelmezésében betöltött szerepüket. A célom az, hogy megmutassam: ezek a – részben a hermeneutikai filozófia tradíciójából, részben pedig a posztpozitivista tudományfilozófiai diskurzusból eredeztethető – fogalmak igen hatékonyan alkalmazhatók valós életszituációk értelmezésére napjaink technicizált világában is. Dolgozatomban építek Don Ihde, Hubert Dreyfus, Michel Foucault, Michel de Certeau, Bruno Latour és Lawrence Lessig elgondolásaira, és ezeket a számítógépek megismerésben, megértésben és társadalmi struktúrák alkotásában betöltött szerepének elemzésére alkalmazom.

Első esettanulmányombana számítógépes nyelvnek a fejlesztők közötti megértés és a hatalmi viszonyok közvetítésében játszott szerepét elemzem. A tanulmányban történeti forrásokra, cikkekre, levelezési lista archívumokra, kvantitatív statisztikai elemzésekre támaszkodva rekonstruálom az Ada számítógépes nyelv fejlesztőinek gondolkodását a hidegháború kontextusában, és ezt összehasonlítom a Python szabad forráskódú nyelv fejlesztőinek a megközelítésével. Célom az, hogy bemutassam: a két különböző programnyelv eltérő hatalmi viszonyokat, megértési stratégiákat,

tevékenységformákat és értékeket kódol, és ezzel magyarázható, hogy történetileg más-más társadalmi kontextusokban bizonyultak sikeresnek.

Második esettanulmányomban szoftverfejlesztőkkel folytatott interjúkra, és egy hibakeresésről szóló első személyű rekonstrukcióra támaszkodva azt tervezem bemutatni, hogy a számítógépes kódok olvasása a klasszikus szövegek megértéséhez hasonlítható értelmezési (hermeneutikai) tevékenység, amely egyrészt az eredeti író megértési horizontjának a rekonstrukcióját igényli, egyúttal azonban az értelmező horizontján belüli kreatív rekontextualizáció is. A programozók által használt számítógépes nyelvben kódolt fogalmi „absztrakciókat", mint megértési és hatalmi stratégiákat vizsgálom, továbbá párhuzamot építek ki az értelmezési szabadság problémája és a kód különféle szituációkban történő újrahasznosíthatósága között. Célom az, hogy bemutassam: a klasszikus filozófiai és a központi jelentőségű technológiai kérdések között lehet és érdemes párhuzamokat vonni, mert ez mind a filozófusok, mind pedig a technológusok számára mélyebb belátást tesz lehetővé szakmájuk társadalmi jelentőségét illetően.

Harmadik esettanulmányomban Michel de Certeau elméletét követve bevezetem a felhasználói appropriáció fogalmát, és a Netscape/Mozilla webböngésző létrejöttének történetén illusztrálom ennek kortárs relevanciáját. Az appropriáció itt azt a folyamatot jelenti, ahogyan a felhasználók birtokba veszik, testreszabják, beintegrálják életükbe a technológiai eszközöket, gyakran a tervezők által egyáltalán előre nem látott módokon értelmezve újra annak funkcióját és esztétikai világát. Elgondolásom szerint itt az újraértelmezés a fejlesztők és felhasználók közötti hatalmi viszonyok újradefiniálásával jár, ezek az epizódok pedig jól tettenérhetőek a Netscape/Mozilla projekt egyes fordulópontjaiban. Forrásként publikusan elérhető archív adatokra (levelezési listák, projekt tervek, történetek) támaszkodom.

1.3. Abstract

In the thesis, I propose a novel way to look upon software. Revitalizing the perspective of the hermeneutic tradition of philosophy, I argue that *we should look upon software as a region of the phenomenal world of our contemporary life: a shared field of meanings, experiences, interactions, and skillful practices*. This perspective draws attention to questions such as: What idealizations of human interaction do we build into our systems? How do these ideals transform our interactions and experiences within the world of software? How can we resist against the values, policies, and social biases that are embodied by them? How can we grasp the immense *social* complexity of the systems we've created?

In a certain sense, this thesis turns the original question of Artificial Intelligence – whether symbol-processing systems can "genuinely" think, experience, and interact with humans – upside down. The question now becomes: how do *we* experience symbol-processing systems, how do *we* think and interact with them?

To address these issues, I introduce the concepts *hermeneutic practice*, *power field*, and *user appropriation* in the contexts of use and development of info-communication technologies. Instead of giving abstract definitions, I make these concepts meaningful through showing their relevance in detailed examples and case studies.

First, I assess the role of programming language in *mediating understanding and power relations* between developers. Drawing upon a variety of historical sources, mailing lists, and quantified data, I reconstruct the language designers' reflection upon these questions in the case of the Ada programming language in the social context of the Cold War, and I compare it with the approach of the open-source development community of the Python language. My goal is to demonstrate that the two programming languages encode different power relations, understanding strategies, practices and values, and this is why they were successful in different social contexts.

In my second case study, I draw upon a semi-directed interview and a first-person perspective reconstruction of a debugging situation to show that reading computer code is an *interpretation practice* comparable to the reading of classical texts. On the one hand, it involves the reconstruction of the original writer's horizon – circumstances, intentions, beliefs and meanings –, but on the other, it is also a creative recontextualization of the text within the new horizon of the interpreter. I'm analyzing conceptual "abstractions" as strategies for sharing understanding and establishing control over situations.

Third, following Michel de Certeau, I introduce the concept of *user appropriation*, and use it to analyze the shifts of power during the development of the Netscape/Mozilla web browser. *Appropriation* refers here to the process in which the technical device is possessed, customized, and integrated into the life of its users, often by reinterpreting its function and aesthetics in fully unanticipated ways. Reinterpretation redefines the power relations between users and developers, and such episodes can be identified in the Netscape/Mozilla project.

1.4. Acknowledgements

I'd like to express my gratitude here toward my tutor Tihamér Margitay, who encouraged me to carry forth with this research through the numerous detours it took me to arrive here. I'm also thankful for the unwavering support of my mentor, Márta Fehér, and for István Zentai for the provoking philosophical questions he brought up while we were doing cross-country mountain biking. Imre Hronszky and László Ropolyi influenced me significantly by exposing me to ideas and literature in history, sociology, and philosophy of technology. I'm also glad to have taken part in Csaba Pléh's seminars on Cognitive Psychology, which gave me a unique opportunity to peek into the state-of-the-art in the research of human cognition. Gábor Zemplén, János Tanács and Benedek Láng proved to be astute and untiring critics, without whose clever comments this work wouldn't have taken its present form. The discussions with my colleagues Gergely Timár, Levente Török and Gergely Kertész on the methods of software development and on the societal relevance of software have proven invaluable in forming my philosophical conceptions. I also thank the anonymous interviewees appearing in the second and third chapter for having shared their experiences with me.

I'm grateful for the opportunity to take part in the workshop discussions of the Department of the Philosophy and History of Science, which gave me important feedback on the consistency and relevance of my ideas. I thank Lars Risan for having organized the FLOSS Workshop in Helsinki, and the participants of the workshop for giving me very relevant comments and suggestions on the first part of this work.

Finally, above all, I'd like to thank my family and Ada Fekete for their continuous support and encouragement, without which I couldn't have carried out this work.

During the research, I was a member of PhD School of the History of Science and Technology at the Budapest University of Technology and Economics (BME), Department of the Philosophy and History of Science (PHS). At the stage of its completion, I enjoyed the financial support of the BME-HAS Béla Julesz Cognitive Science Research Group, and the Jedlik Ányos Research Grant (NKTH KPI NKFP6-00107/2005). Most of this research, however, was carried out under the support of the BME-HAS Research Group for the History and Philosophy of Science.

1.5. Sources of influence

The book that made me embark upon this path of research is definitely *Understanding Computers and Cognition* by Terry Winograd and Fernando Flores (1987) (nowadays leading theorists in Human-Computer Interaction and Business Process Reengineering). This book was an argumentation against the then-prevailing conceptions of artificial intelligence, which derives its core arguments partly from the premises of the hermeneutic tradition (Heidegger, Gadamer and Dreyfus), and partly from Maturana and Varela's system-theoretic perspective on life and evolution. I was astonished by the preciseness of their diagnoses and the validity of their predictions. I've started to analyze their arguments and I soon found myself drawn into the field of philosophical and sociological critiques of classical cognitivism. It was particularly Dreyfus, Agre, and Suchman, who made me receptive toward problems of *situatedness* and *skillful practices*. I've approached the classical works of Heidegger and Gadamer from this interpretative backround, and I've read them as an immense source of inspiration for opening new perspectives on the problems I've encountered in my daily routine as a software developer.

This research led to a publication tracing the intellectual history of the most important pro and contra arguments about the possibility of artificial intelligence (Binzberger, 2006). It was this process, in which I came to realize the power of the philosophical concepts of the hermeneutic tradition, and at the same time, I came to see the almost total neglect of the social contexts of computing by philosophers. These interpretators – standing mostly isolated from mainstream analytical or continental philosophy – have demonstrated how can philosophy be made relevant in contemporary debates: how can one engage the classical ideas to bring the phenomena of our concerns under new illumination.

The other decisive impulse for my research came from postpositivist philosophy of science, particularly from the works of Joseph Rouse, Don Ihde, Patrick A. Heelan, and Bruno Latour. My third important source of inspiration is American history and philosophy of technology: the critical philosophical investigations of Andrew Feenberg, and the historical reconstructions of organization and computing in the era of the Cold War by Paul N. Edwards, Thomas P. Misa, Peter Galison, and David Holloway.

Last, but not least, I owe a great deal to the many theorists of software development, listed in the bibliography – David Parnas, Walt Scacchi, Andrew Hunt, Tom DeMarco and Timothy Lister, to name just a few –, and the hundreds of open-source contributors in the mailing list debates I've read while preparing my case studies. They've manifested a great deal of humanity in issues regarding the management of software development, and a deep understanding about the power relations involved.

2. Introduction

2.1. Problem setting

What is the philosophical status of software? Shall we think of it as an independent ontological level, a physically embodied technological device, or a pattern of functioning, existing only in the eye of the beholder? Much ink has been spilled over these issues in philosophical debates over functionalism and the possibility of Artificial Intelligence. What all the participants in these debates have in common is that they all wanted to give *theoretical* answers to these questions, abstracting away from the realities of existing systems and the situations of their uses.

In the meanwhile, software has turned out to be the single most pervasive medium of human culture and understanding. It has thoroughly transformed the ways in which we perceive the world and ourselves, the patterns in which we interact with each other and how we influence the interactions of others. The flexible control architectures, supplemented with a rich set of causally effective interfaces opening to our physical environment enable software systems to embody and enforce values, policies, aesthetics, and ideologies on a previously unprecedented scale. This potential of *re-presenting* the absent designers and operators seems to confuse the distinction between humans and non-humans not only in our philosophical descriptions, but our everyday dealings with technology as well. The abovementioned philosophical debates nevertheless did not result in a conceptual framework and a methodology with which we could address these issues. Focusing on the software-mind analogy and on the theoretical question whether a computer can be intelligent *in itself*, they've completely lost sight of the fact that *first and foremost, software is never to be found "in itself": software is something that people do.* In the 21st century, it is still humans who are engaged in creating, using, tailoring, maintaining, interpreting, transforming, and quite literally living in software, and it is also humans who are affected by the side effects of this technology.

In this thesis, I propose a novel way to look upon software. Revitalizing the perspective of the hermeneutic tradition of philosophy, I argue that *we should look upon software as a region of the phenomenal world of our contemporary life: a shared field of meanings, experiences, interactions, and skillful practices.* This perspective draws attention to questions such as: What idealizations of human interaction do we build into our systems? How do these ideals transform our interactions and experiences within the world of software? How can we resist against the values, policies, and social biases that are embodied by them? How can we grasp the immense *social* complexity of the systems we've created?

In a certain sense, this thesis turns the original question of Artificial Intelligence – whether symbol-processing systems can "genuinely" think, experience, and interact with humans – upside down. The question now becomes: how do *we* experience symbol-processing systems, how do *we* think and interact with them?

To address these issues, I introduce the concepts *hermeneutic practice*, *power field*, and *user appropriation* in the contexts of use and development of info-communication technologies. Instead of giving abstract definitions, I make these concepts meaningful through showing their relevance in detailed examples and case studies.

First, I assess the role of programming language in *mediating understanding and power relations* between developers. Drawing upon a variety of historical sources, mailing lists, and quantified data, I reconstruct the language designers' reflection upon these questions in the case of the Ada programming language in the social context of the Cold War, and I compare it with the approach of the open-source development community of the Python language. My goal is to demonstrate that the two programming languages encode different power relations, understanding strategies, practices and values, and this is why they were successful in different social contexts.

In my second case study, I draw upon a semi-directed interview and a first-person perspective reconstruction of a debugging situation to show that reading computer code is an *interpretation practice* comparable to the reading of classical texts. On the one hand, it involves the reconstruction of the original writer's horizon – circumstances, intentions, beliefs and meanings –, but on the other, it is also a creative recontextualization of the text within the new horizon of the interpreter. I'm analyzing conceptual "abstractions" as strategies for sharing understanding and establishing control over situations.

Third, following Michel de Certeau, I introduce the concept of *user appropriation*, and use it to analyze the shifts of power during the development of the Netscape/Mozilla web browser. *Appropriation* refers here to the process in which the technical device is possessed, customized, and integrated into the life of its users, often by reinterpreting its function and aesthetics in fully unanticipated ways. Reinterpretation redefines the power relations between users and developers, and such episodes can be identified in the Netscape/Mozilla project.

The method of my analysis involves *hermeneutics* on two levels. On the first level, we should recognize the importance of the activities in which technologically situated actors engage themselves to understand each other: the importance of hermeneutic practices *within* the realm of technology. On the second level, *the analytic method itself* is a hermeneutic one: we reconstruct the meaning of the technical artifacts and rationales while trying to bring closer to the reader the cultural horizon, in which they were originally meaningful.

2.2. Why do we need a *hermeneutic* philosophical approach to understand software?

Why is it *hermeneutic* philosophy, which is *particularly* relevant in understanding software? A short answer would be that it is exactly this tradition in which it is most straightforward to say that *software has to be understood by the role it plays and the meaning it has in the experiential contexts of its development and its use.* The meaning of software is not exhausted by objectivist, disembodied representations of it as a set of bytes, computing states, or data flows. The meaning of software depends on the interpretations of

the social and physical world that are built into it, and it also depends on how it gets interpreted in the subsequent interactions of its users and developers. There is no privileged narrative defining what the particular software "is": there is always a multiplicity of such narratives.

Hermeneutic philosophy of technology is also very relevant to our purposes because of its *special focus on skills and practices, on the embodiment and situatedness of experience*, standing in contrast to the linguistic orientation of contemporary mainstream philosophy. Hermeneutics – particularly after Heidegger – does not delimit its scope to what is expressible in natural or scientific language. *Interpretation is not an explication in a different language*: it happens already in the skillful interaction with the things at hand. It does not mean taking up arbitrary (theoretic) assumptions with respect to the object in focus: the unarticulated background of moods, skills, and circumstances, against which interpretation takes place, cannot be chosen at will. Since the interactions with the software, the practices of development and use are often not symbolically articulated[1], this philosophical approach seems to suit well our domain of interest.

The interpretations built into the software open up and structure the field of possible interactions with the system, thereby constraining the range if its possible interpretations. In some cases, these constraints impose rigid limits on what is doable and thinkable within the confines of the system, whereas in others, they appear only as subtle implicit biases, slightly favoring certain interpretations over others. This means that software constitutes a *flexible and often invisible power field*, which influences the users' possibilities of action, experience, and thinking – sometimes without the chance of informed consent or even dissent. Nevertheless, creative reinterpretation can contest and disrupt established power patterns mediated by the software by discovering and working out new paths of possible interactions.

Beside hermeneutics, the methodology of Social Construction of Technology (SCOT) also offers an interpretativist stance on technology (Bijker & Pinch, 1987), and the notion of "interpretative flexibility" is widely shared in the field of Science and Technology Studies. Why do I insist upon the relevance of hermeneutic philosophy, instead of drawing upon the well-established conceptual framework of SCOT? I answer this question at length in the second chapter, where I analyze the differences between the two approaches, and I argue that the hermeneutic tradition – because of its thematic focus on the experiential aspect of technologies – offers more intellectual resources for *social critique* than SCOT, which rules out critique as an objective of scholarly discourse given its methodological commitment toward *symmetry*. The most important critical insight, which follows from the hermeneutic approach, but would be hard to describe within the SCOT paradigm is that user experi-

---

1 This is particularly true for graphical and tangible interfaces (Dourish, 2001), but I will argue later that the use of command line interfaces also draws upon a rich, unarticulated background of skills and cultural orientations.

ences, interpretations and power relations are closely interrelated: *interpretation means taking up a position in a power field, thereby affirming or contesting its existence.*

I work out and corroborate these theses through *reconstructing the situated perspectives and the unarticulated backgrounds of developers and users in various use-situations over a wide range of different software systems:* computer language compilers, web development frameworks, and various commercial and open-source projects. These descriptions are aimed at demonstrating the inherent ambiguity of possible interpretations and the fantastic diversity of uses, in order to dispel with the myth of the "software in itself" – the software as if it existed and had "functions" independently from us.

2.3. Theses

The dissertation applies classical philosophical approaches to contemporary empirical material, the practice of software development. This involves the extensive reinterpretation, adaptation and critique of the classical conceptions. I think the most important original result of this work is the link itself that has been thus created between the two fields. My detailed theses are the following.

T1. The everyday practices of writing, deciphering, using, adapting, and appropriating software artifacts by users and developers can be understood as *hermeneutic practices*. I'm defining this expression in order to bring emphasis on the following meaning components of the concept "hermeneutic":

a.) *Hermeneutic practice is a process of interpretation that unfolds itself in skillful action* rather than explicit, linguistic articulations. The resulting understanding does not mean holding and manipulating explicit assumptions or symbols, but rather *the capability of orienting ourselves – coping – within a shared lifeworld.*

b.) All practical engagement involves a hermeneutic component, because it already involves perceiving the world as purposeful, as one that offers affordances, as one that can be translated into a *field of possibilities for action and existence*.

c.) Hermeneutic practice is *situated* and *embodied*, that is, relative to the bodily-perceptual skills of the interpreter, and embedded in the material and social concreteness of the situation. The meaning of objects and symbols has to be reflected within the local context of other objects, social circumstances, skills, etc.

d.) Hermeneutic practice presupposes having already taken up a position within *a holistic horizon of meaning, constituted by a shared lifeworld and shared traditions of interpretation*. Understanding

even a single symbol of the code involves understanding a precisely non-delimitable set of other symbols, skills, traditions, and situations.

e.) Understanding as a process is inevitably circular, because we don't have any other means to assess the relative value of our interpretation than to engage in further interpretation, based on whatever understanding we've already arrived at so far. In the case of interpersonal communication, this circularity takes the form of asking back and then reinterpreting what has been said so far; in the case of interpreting texts, it takes the form of taking a fragment that is already believed to be understood and then reassessing its relations within the totality of the text; and in the case of interpreting software, it takes the form of actively interacting and experimenting with it.

T2. *Software, as a medium of hermeneutic practice, can determine the field of practical interpretations that can be carried out within it:* the field of possibilities of understanding, experience and action for those who interact with it. Especially in the case of computer language, this field influences the potential dimensions of conceptual abstractions that can be applied during the interactions with the software.

T3. *Software and programming language can constitute a field of power.* By transforming the space of possible interactions that can be carried out with it, developers can restrict the multiplicity of its possible interpretations; they can inscribe preferred patterns of usage in the software as privileged interpretations. Software thus can be leveraged to maintain more traditional forms of power relationships, to normalize practices, and to serve as *obligatory passage points* (Latour, 1987).

T4. *During the process of user appropriation, software – just like texts and other artifacts – can always be subject to unexpected reinterpretations that challenge the preferred patterns of usage instilled by its developers.* The importance of open-source software development in these appropriation processes is significant, both because the field of possible reinterpretations is wider, and also because these interpretations are argued and contested in front of public scrutiny.

2.4. Structure

The structure of the work is organized around individual cases, which bring up a set of issues that are connected to more than one of the outlined theses. It consists of three introductory and three main chapters. The first main chapter assesses *the role of computer language in mediating understanding and power relations* between developers by comparing the differences of language use in the Ada and Python development communities. It also tries to judge the relevance of the ideological self-assessment of Python developers by confronting it with quantitative data gathered from mailing lists, and by a qualita-

tive assessment of the communication patterns of a Python-based open-source development community, Turbogears.org.

The second main chapter is aimed at showing that *reading and debugging computer code is an interpretative practice* comparable to the reading of classical texts. On the one hand, it involves the reconstruction of the original writer's horizon – her circumstances, intentions, beliefs and meanings –, but it is also a creative recontextualization of the text within the new horizon of the interpreter as well. Fixing a bug also involves reconstructing the expected circumstances of future usage, and making judgments based on these reconstructions.

The third main chapter analyzes the *role of creative user appropriation in contesting established power relations* that are imposed on the users of the software by the developers. User appropriation is analyzed as the reinterpretation of the technical artifact, and this process is exemplified by assessing the shifts in the balance of power in the history of the Netscape Mozilla project, based on a reconstruction from publicly available sources.

2.5. Methods

My choice of the level of detail and depth of analysis was guided by the hope that this work will prove to be interesting for software developers and philosophers alike. This intention sometimes forced me to make compromises between preciseness and comprehensibility, to strike a balance between the conceptual meticulousness of the philosophical reconstructions and the technical details of the case studies in order to produce self-contained descriptions that are comprehensible for a wider range of audience.

The consequences of this decision are particularly discernible in the choice of terminology. Instead of using Heidegger's technically precise, but obscure standard terminology ("Dasein", "In-der-Welt-sein", "Sein-können" ...), I tried to paraphrase or circumscribe them with similar, but more commonly understandable metaphors from the tradition, such as "lifeworld", "horizon", "the world as it opens up before us", "skills of orientation", and so on. I am aware that this decision sacrifices terminological exactness for the sake of comprehensibility, but I maintain that these terms are *precise enough for our purpose*, namely, for opening up new perspectives on understanding software[2].

Following Gadamer's theory of meaning – according to which the meaning of a word lies in the situations of its application[3] – I use three mutually reinforcing strategies for clarifying my concepts.

---

2 For example, using the words "programmer" and "program" is already loaded by the metaphysical subject-object dualism criticized all throughout the tradition, but except for the theoretical discussions where this very dualism is at stake, I think it would be awkward to depart from the established usage of these words.

3 Parallels could be drawn here between Gadamer's concept of "application" and the late Wittgenstein's parable of "language games" based on their *infinitistic* conception of meaning, as analyzed by (Bloor, 1983), but such a comparison would reach beyond the scope of this introduction.

In certain cases, I give explicit definitions, thereby applying the concept in the idealized situation of the "everyday use" of language. In other cases, I introduce terms by showing their relations to other concepts of the tradition, thereby situating them in its interpretative horizon. In many cases however, *it is itself the application of the concept in the context of a case study, in which it becomes meaningful.*

This latter sentence amounts to admitting that *the meanings of concepts are not final*: they are subject to drifts and reinterpretations as they are being applied in new situations. Of course, I tried to do my best to maintain the consistency and coherence of concepts throughout the thesis, but this inherent open-endedness of meaning has well argued theoretical reasons within the philosophical tradition. I chose a strategy that avoids shifting to the meta-theoretical level of discourse: I shall show *at the subject level of our investigation* that these shifts in meaning are indeed very common even in the use of formal language.

The historical reconstructions in the text follow a method called "double hermeneutics", which is understood to be hermeneutic on two levels (Heelan, 1989; 1997; Rouse, 1987: 59). On the first level, it focuses on the interpretative activities in which the technologically situated actors are engaged in order to understand themselves and each other. Since this interpretative activity is not exhausted by the explicit reflections of the actors, but takes place rather in patterns of usage and work, I denote them with the term *hermeneutic practices*. On the second level, our method itself is a hermeneutic one: we reconstruct the meaning of the technical artifacts and rationales as they result from the actors' hermeneutic practices, by the role they play in their lifeworld.

During this process, we first collect publicly available traces of these hermeneutic practices – explicit reflections, design rationales, and argument fragments revealing worldviews and valuations –, while trying to bring our perspective closer to the cultural horizon, in which they were originally meaningful. In a few cases, I relied on experiences gathered in interviews from sources that were left anonymous. Afterwards, we attempt to give a description of this system of meanings in the language of our "expected reader", in order to transform her horizon, to bring her perspective closer to that of the original participants. This process involves the fusion of at least three horizons – that of the actor, the theorist, and the reader –, but in each case study – arguing for the existence of multiple perspectives on software –, I compare the angle of at least two actors, so in each case there are actually four perspectives in play.

Naturally, such an interpretative situation could result in a failure of understanding at each of the levels, but things are not as complex as they may first sound: the situation is not more complicated than reading an analysis of a debate, or listening to a dialogue in a drama. The difference is that in these latter cases, we might naively assume that all of the participants share the "same" horizon, whereas in the former, we are forced to notice the differences.

These considerations draw attention to an even more serious point of possible methodological criticism against this work. I might be charged with the claim that my philosophical agenda permeates the selection and the presentation of my historical data: my "hermeneutic glasses" distort my percep-

tion to see only the confirming instances of the philosophical theory I'm putting forward. I take up a theoretical orientation in the form of a "tradition", which I neither theoretically defend, nor fully articulate, only to give descriptions in it; and then I expect that the validity, or the usefulness of these descriptions somehow "confirm" the tradition itself.

At first glance, this does not seem to be a valid analytic form of reasoning. This methodological issue has been widely reflected upon in hermeneutic philosophy: this is the problem of the "hermeneutic circle". Interpretation cannot be carried out with a "blank mind"; the interpreter has to take up a position in the horizon of a tradition, a position which he cannot "confirm" or "disconfirm" directly. The best he can do is to show that the interpretation can be fully carried out in a fashion that is consistent and adequate according to the standards of that tradition. The "theory-ladenness of data" argument can be mounted not only against my work, but also against *any* historiography, or any theoretical articulation in general.

3. The problem horizon

The core topic of my thesis, the question of how understanding and power relations are mediated by technological artifacts, belongs among the central issues of Philosophy of Technology. Questions about whether technologies exert undesirable or positive influence on our established systems of control, whether technology can or should be politically controlled have been standing in the focus of heated debates between philosophers for over half a century now. Maybe this is the reason why philosophers of technology generally define their positions in relation to the classical theorists of technology and society – Marx, Heidegger, Jaspers, Ellul and the Frankfurt School, particularly Marcuse – even if they are outspoken critics of their legacy. These classics had profound and lasting influence on contemporary philosophical thinking, and their complex and often highly provocative perspectives offer a wealth of intellectual resources to address contemporary problems.

Yet, it seems that in Philosophy of Technology a new generation is emerging, who are trying to break these strong ties with the tradition through constructively extending, reinterpreting and criticizing the classical programs. This generation is characterized by a deep understanding of the classics, and a creative impulse to adapt their ideas to the new situations. Among the most prominent theorists of this new generation we can find Hubert Dreyfus, Don Ihde, Andrew Feenberg, Joseph Rouse, and more recently, Peter-Paul Verbeek, to name just a few.

There exists also a heterogenous tradition of philosophical reflection on technology, often overlooked by academic philosophers: that is, the self-reflection of technologists; which Carl Mitcham calls "Engineering Philosophy of Technology" (Mitcham, 1994: 19). Philosophical and sociological theories sometimes picture technologists as opportunistic positivists, but I've found that this is far from being true. In all ranks of the technological order, we can find people who are sensitive to questions of democracy, environment, and the impact of technology on the work and living conditions of its users, and these people are setting precedents with their deeds and designs for many future generations of technologists. Throughout all my case studies, I've laid special emphasis on taking into account the situated perspectives of designers and engineers, not solely because of their philosophical relevance, but also because according to my thesis, their engineering philosophies are reflected in the design of the technological artifacts which surround us.

3.1. Hermeneutics and philosophy of Artificial Intelligence: Hubert Dreyfus

Dreyfus's famous work, *What computers cannot do* (1967), draws upon Heidegger's and Merleau-Ponty's phenomenology in criticizing the then-prevailing philosophical conceptions of Artificial Intelligence research. He adapts the phenomenologists' critique of cartesinanism against the dominant symbol-processing paradigm of AI research. His core argument is that the decontextualized, propositional, symbolically articulable form of "knowledge", which AI researchers are trying to build into computers, is just the tip of the iceberg: human cognition is rooted in the immensely rich background of

unarticulated, bodily skills, social habitus (Bordieu), and the structured use of the environment itself. The "representations" we use to find our way around in the world are not the universal rules and symbolic states AI researchers are after, but rather hard-to-explicate skills of recognizing relevance, similarity, and analogy. Humans are not passive information processors either: they actively seek relevant experience, and they learn skills by practicing and imitating others.

Looking back after 40 years, we can see now that most of his predictions have proven to be correct. The problem of intelligence turned out to be the hardest nut scientists ever tried to crack. Despite heavy hype and extended media presence, humanoid AI is not among the many things that have contributed significantly to the immense changes in the human life experience by the beginning of the 21st century[4]. Computers are everywhere now, and computing power surpasses that of the 60-ies billions of times, yet even market-leader electronics giant Sony had to shut down its humanoid robot research branch in 2005 because of its limited success and low profitability. Finally, the famous Turing test has turned out to be relatively inconsequential with respect to the nature of human intelligence, since simple people can be relatively easily tricked into mistaking a computer for a human, and even sophisticated people seem to find enjoyment in playing against relatively crude "AI" players for an extended time.

A bit paradoxically, the success of Dreyfus's critique provides an indirect *empirical* proof for the phenomenologists' vision on human experience and thinking. His method exhibits a deeply empirical character also in a different sense. In contrast to Searle, Block, Dennett, Putnam, Clark, and many others taking position in the debates over the possibility of AI, he visited the laboratory of the AI researchers, looked into their systems, analyzed their self-perception based on their research proposals and reports, and followed the public reception of their pronouncements – quite unlike philosophers, but more like a Sociology of Science practitioner, like Collins (1990) does. And in contrast to others, he took the intellectual risk of swimming against the tide, making clear, testable, long-term predictions against mainstream hype, arguing not for the relatively inconsequential *philosophical* inconceiveability, but the *practical* unfeasibility of symbolic AI way before anybody else started to express doubts against the optimistic general outlook.

Empirical proof and *practical* conclusions from a *transcendental* philosophy? It doesn't seem so far-fetched if we keep in mind that AI research has lots of unreflected philosophical assumptions about human experience, inherited from the rationalistic tradition through the work of Bertrand Russell, Herbert Simon, and many others; and hermeneutic phenomenology is concerned exactly with the questioning of these philosophical assumptions. Should the intelligent machine have proven easy to

---

[4] Dreyfus makes an explicit distinction between general-purpose human-like AI and limited-domain rule-based reasoning systems, which he considers quite successful, but never intended to be "intelligent" in the genuine sense.

build, based on classical, symbol-processing AI, then the analyses of hermeneutical phenomenology would have proven to be of no or little relevance in the face of the rationalistic tradition[5]. Since this has not turned out to be the case, I suggest that we should pay more attention to their results. Furthermore, (hermeneutic) phenomenology is not, in its original intentions, antiscientific *per se*. Heidegger, Merleau-Ponty and Dreyfus all look upon certain empirical research programs in biology, neurology and AI as fundamentally reconcilable with their approach[6].

In a certain sense, my thesis inverts Dreyfus's concern. Accepting that symbolic systems are so different from our thinking, because the latter is rooted in a rich background of culturally situated skills, the question becomes now: How do we, humans experience and interact with symbol processing systems? What is the impact of using symbol processing systems on our ways of thinking? I take up this issue explicitly in the first and the second case study, where I shall try to shed light on the background of understanding, arguing that we perceive formal symbols as meaningful, embedded in the context of the use and development of the software system.

3.2. Techno-phenomenology and embodiment: Don Ihde

Don Ihde has embarked upon a much less confrontative path to extend phenomenology into the domain of technoscience. In *Program One. A Phenomenology of Technics (1990)* and *Expanding Hermeneutics: Visualism in Science (1999),* he brings into attention the role of imaging technologies in scientific and everyday experience. Microscopes, MRI machines and TV news all blur the boundary between subject and object, observer and observed: they transform the perceptual field of the user, thereby bringing him into a *deceptive immediacy* with the object – even if the object is constructed through sophisticated manipulations, computations, or elaborate staging in the media machinery.

"Perceivability is polymorphic, and is always both bodily and cultural" (Ihde, 1999: 97). Every act of perception involves our bodily-sensual skills, our culturally acquired background of interpretation, and the cultural heritage built into the imaging apparatus. Perception is not a passive reception of images, but an acquired, technological-bodily capability of constructing (visual) inscriptions and learning how to see "through" them. In the case of microscopic observations, for example, the

---

5 (Collins, 1990: 8) makes a similar point with respect to the prospect of the Sociology of Science. "If there can be machines that act indistinguishably from us, then the philosophical distinctions between action and behavior, and the argument about the peculiar nature of human rule-guided action, will turn out, after all, to be of no significance for the prospects of a *science* of society. The pigeons of philosophy of social science are coming home to roost in the intelligent computer."

6 Incidentally, Heidegger was influenced by the very same biological theorist, Jakob von Uexküll (Heidegger, 2000), who is often cited as the precursor of the anti-representationalist "nouvelle AI" approach (Brooks, 1991), which is one of the contenders of classical symbol-processing AI.

breakthrough came with the discovery of the staining process through aniline dyes, which caused the important biological features to visually "pop out" from the otherwise transparent sample (Ihde, 1999: 166). Then the staining and observation technique had to be refined and tuned in order to see what is relevant for the biological explanations.

No matter if philosophers of science consider electrons and neurons as "theoretical constructs", or news commentators the mediatized wars of the 21st century as "virtual reality", the solid belief in their realness is founded upon the imaging technologies which make them appear within the range of our senses. Of course, our reliance on imaging technologies involves the risk of being misled by "artifacts" – objects, whose perceived existence owes only to a systematic distorsion in the apparatus. Nevertheless, according to Ihde, the greatest risk of imaging technologies lies not in false perception, but in their essential transformative nature. They construct and bring close otherwise unperceivable objects, but at the same time, they occlude other, possibly relevant dimensions. There is always a trade-off between the visible and the invisible, but the ubiquitousness of imaging technologies and the richness of the inscriptions produced can lure us into believing that we can have a "total world picture", a "veridic representation" of everything there is to be experienced. The danger is that we are tempted to forget that all perception takes place within a perspective constructed by the body, culture, and technology.

Furthermore, these are not exclusive properties of visual images. We can also "see through" texts, diagrams, datasets, and – as I shall illustrate – program codes. While reading, normally we don't see the letters: we "see through" them, and focus our attention on what is described in the text. Ihde calls this experience *hermeneutic intentional relation* (Ihde, 2003: 517): the case when perception does not take place in the usual dimensions of our bodily senses, but in a space opened up by interpretation. My thesis is connected to this notion insofar as it demonstrates how the symbols of the program code are perceived as meaningful by the programmers, and how this perception is influenced by the design of the programming language and environment. I shall argue in the second case study that what programmers can "see through" the code future are conventions, use-situations, perceptual analogues, and much more.

3.3. Power and resistance: de Certeau and Foucault

I first encountered Foucault's rich rendering of power in Joseph Rouse's book *Knowledge and Power: Toward a Political Philosophy of Science* (1987), and my understanding of it is still influenced by his interpretation. What Foucault has left behind is actually not a "theory of power", but an approach to analyze practices and devices as they structure the field of other practices, as they are inscribed on bodies, and as they construct objects of knowledge. Rouse starts from Foucault's analyses and interprets techno-science practices in their light. Common laboratory practices, like spatial isolation, controlled interaction, nominal classification, description, explanation, surveillance and documentation, can all

be viewed as *techniques of power*: strategies aimed at controlling the experimental situation and extending the laboratory into the real world (Rouse, 1987: 217). As I shall argue in the second and third case study, code-producing laboratories show similar features. On the one hand, they are trying to stabilize and bring their internal code-producing practices under control, and on the other hand, they strive for influence over the situations of the software's usage in the real world.

Extending the laboratory into the world has an important impact on our thinking. It reinforces the (metaphysical) conception of the world as consisting of isolatable objects and relations that are to be controlled similarly to the experimental systems of the laboratory. It is this conception of the world as a system of calculable resources, according to Rouse, which paves the way toward such initiatives like replacing whole biological ecosystems with controlled, genetically engineered artificial systems in order to reliably reproduce the results of laboratory research at large.

A similar argument can be told about software. Software replaces preexisting social ecosystems with new ones, according to the rationales and constraints built into it by the designers. Conceiving the social world of users as configurable resources, "clickstreams" or "data mines" for robots to "harvest" are examples of extending the user model – built into the code – into the real world: the consequences of such thinking are worth exploring.

The central difference between my approach and Rouse's is that in my opinion, he does not put enough emphasis in his examples on another central notion of Foucault. According to Foucault, the exercise of power always happens in the face of *resistance:* power is constituted by an antagonism of forces, and in order to understand its working, we should trace the various forms of resistance in our historical reconstructions (Foucault, 1982). In contrast, Rouse's experimental objects and technicized societies do *not* put up resistance against intervention and control: they passively endure manipulation, and the only resistance they offer is merely a sort of friction against the inevitable progress of the extension of techno-science.

Michel De Certeau (1984) follows an explicitly foucauldian program in his magnum opus, *The Practice of Everyday Life (1984)* in order to dispel with the modernist myth of the consumer as a passive, unresisting receptacles of mass-produced products. He traces the everyday use-situations of various technical objects – buildings, cities, kitchen appliances, texts – as forms of resistance against the preferences and constraints imposed by producers on the possible ways of experiencing their products. He analyzes the power relations imposed by modern systems of technological production from the *perspective of the subjugated.*

His project is a *phenomenology of everydayness* in the Heideggerian sense insofar as it builds upon analyses of everyday experiences, written from the perspective of the experiencer. But de Certeau's *everydayness [quotidien]* is the total opposite of Heidegger's *everydayness [Alltäglichkeit]* as the averaged-out, unauthentic mode of being, which is governed by the moods, customs and commonplaces of the *"others"*. In his passionate first-person perspective portrayals, the *everyday* becomes an incessant stream of unique experiences in the playful art of turning the average into something individual and exclusive.

His everydayness is not a "surface" that conceals the depths behind its superficiality; rather it is characterized by a deepness of its own. De Certeau's esteem for the skilled "art of conversation" stands in stark contrast with Heidegger's dismissal of "idle talk", which is characterized by "forgetfulness of being". In his words,

> [T]he rhetoric of ordinary conversation consists of practices which transform "speech situations", verbal productions in which the interlacing of speaking positions weaves an oral fabric without individual owners, creations of a communication that belongs to no one. Conversation is a provisional and collective effect of competence in the art of manipulating "commonplaces" and the inevitability of events in such a way as to make them "habitable." (de Certeau, 1984: xxii)

Nothing can illustrate the contrast between the two thinkers better than this: whereas Heidegger's literati "dwell in language"[7], and their role is to "guard" it from "the ubiquitous and rampantly proliferating impoverishment" (Heidegger, 1998), de Certeau praises the *everyman* who skillfully inhabits the "commonplaces of conversation"! In all three case studies, my analysis follows this latter perspective: the view of the *everyman* enmeshed in technological systems.

3.4. Heidegger

The relation of my work to Heidegger is somewhat ambivalent. On the one hand, the concept of "hermeneutic practice" which I promote, builds on the early Heidegger's concept of hermeneutics as practical, skillful involvement. I'm also influenced by his thinking indirectly, through the work of other philosophers. Almost all philosophers of technology were influenced by hermeneutic phenomenology: even if they are overtly critical with Heidegger, their choice of problems, methods and perspectives still bears the imprint of this influence.

On the other hand, I look critically upon the late Heidegger's sweeping generalizations over the "essence" of technology and technological language. I shall explicate my difficulties with Heidegger in a particular case in the first case study, but I give a short recapitulation of my core concerns here.

I follow Verbeek (2005: 9) in understanding the Heidegger of *The Question Concerning Technology* as presenting a *transcendental* form of argumentation, where he takes the technological artifact and analyzes its condition of possibility by asking back toward the form of thinking (being) that made it possible. Heidegger brings into attention that modern technology is made possible by an epoch of

---

7 "Die Sprache ist das Haus des Seins. In dieser Behausung wohnt der Mensch. Die Denkenden und Dichtenden sind die Wächter dieser Behausung." (Heidegger, 1947)

understanding things as mere resources [*Bestand*], an epoch that is characterized by a totalizing, nihilistic way of objectification, what he names *challenging-forth* [*ausforderung*]. These and other features characterize the *essence* of technology: without these conditions in place, modern technology would be unthinkable. The validity of this claim is illustrated by examples which invoke familiar concepts of the "managerial perspective"[8] or "business talk": human material [*Menschenmaterial*], claim [*Forderung*], stock/resource/reserve [*Bestand*] and so on. But Heidegger does not put forward compelling evidence to show that this is indeed the *only* way in which the technological world is and can be perceived; he does not make a convincing attempt to prove that this is indeed a *necessary* condition of possibility of modern technology.

My argument goes like this: if we want to see whether Heidegger's vision is relevant with respect to modern technology, we should take a look at the thinking and particularly the *language* of technological innovators and key decision-makers. What we find, as I shall argue, that in some cases, Heidegger's analysis is astonishingly correct, and in others, it is completely mistaken. My examples illustrate that it is quite possible to conceive of and successfully develop modern technology while explicitly avoiding understanding things as mere resources.

If this argumentation is correct, than Heidegger's *transcendental* argument does not lead to a *necessary* condition. His concerns either have to be addressed *empirically*, to find the particular historical situation under which they are relevant (like Dreyfus does), otherwise we would need some kind of an explanation for the discrepancy between his vision and the real ideologies motivating technologists. In my thesis, I shall follow the former path.

---

8 This expression was coined by Andrew Feenberg.

4. Hermeneutic practices in software development: the case of Ada and Python

4.1. Introduction

This chapter shows the relevance of hermeneutic philosophy to understand how info-communication technologies frame our contemporary lifeworld. It demonstrates that *the programming languages are the result of collective interpretations of the general lifeworld of programmers, management and political decision-makers. By having been inscribed into the processes of programming language use, this general interpretation permeates the particular practices of understanding that are possible within the language framework.*[9]

I support my argument by contrasting the hermeneutic concerns about the understanding between programmers that stand behind the design of the Ada and the Python programming languages. The technical specification of Ada, its emphasis on achieving seamless communication through rationalistic standardization and technical embodiment of the background of understanding, bears the imprint of the culture of Cold War-era DoD-funded military projects. On the other side, Python is inscribed with the culture of open-ended discussion and self-reflective practices of conventionalization that are characteristic of the FLOSS world.

4.2. Hermeneutics, software technology and understanding

During the process of software development, participants – users, developers, customers – must engage in various *hermeneutic practices* in order to achieve a shared understanding of their software artifacts, source codes, relative valuations of problems, norms, and the general social context of development. These practices can take a range of forms, including producing and using a variety of inscriptions in formal or informal languages (codes, documentations, specifications), or engaging in multi-modal discussions in various media (IRC channels, wiki pages, screencasts). I'd like to show that also the seemingly solitary interaction with human-computer interfaces – like computer language compilers – can be viewed as a kind of hermeneutic practice, aimed at sharing the background assumptions of the interface designer at the very level of skillful actions.

I use the term "hermeneutic practice" instead of "communication", because I want to emphasize that the role of these practices is not exhausted by conveying explicit meaning within a previously given horizon of understanding. These practices are necessary to build out *the horizon itself*, within which communication can take place, and within which explicit symbolic inscriptions – like source codes – can be interpreted. What "gets transmitted" in them is often not an explicit message, but un-

---

[9] This thesis is an application of T2 to the case of computer language.

articulated background assumptions, skills, and orientations, with which particular questions and problems can be approached.

In the case of software, to a great extent, the programming language constitutes the horizon in which programmers articulate and convey their ideas. In my attempt to show how relevant hermeneutic philosophy is to understand how info-communication technologies frame our contemporary life-world, I'll demonstrate that *the programming language framework is itself the result of the collective interpretation of the general lifeworld situation by programmers, management and political decision-makers. By having been inscribed into the processes of programming language use, this general interpretation permeates the particular practices of understanding that are possible within the language framework.*

I shall assess the validity of this statement with two case studies, focused on the hermeneutic concerns of the language designers about the understanding between programmers. These cases have been selected as two influential milestones in the half century-old discourse about what can be considered as "good" programming practice (a good retrospective is in Boehm and Turner, 2004), exemplifying two distinctly opposed ideals as an answer to that question.

I've chosen the Ada programming language as my first example because of the striking similarities between the concerns of its designers and the theories of influential philosophers of the hermeneutic tradition. Besides providing an interesting insight into the practices of Cold War-era DoD-funded military projects, this choice also brings an opportunity to assess the claims of Heidegger and other hermeneutic philosophers about the status of technical language in the light of a contemporary historical example. Furthermore, the insights thus gained are still relevant, since the concepts built into Ada have influenced many contemporary languages, including JAVA.

As my second example, I've chosen to analyze the reflections on one of the most widely known open-source language, Python, and the hermeneutic practices of a Python developer community, www.turbogears.org, because they follow a wholly different strategy to address similar concerns as those of Ada. While the designers of Ada wanted to achieve shared understanding through normalizing programming practice by controlling every detail of the programmer's technical environment, Python designers "factor" these concerns "out" of the language and shift them into the normative disciplinary space of the surrounding discourse. To use Barry Boehm's distinction (1979), Ada focuses on a "restricted view" of practices, whereas Python builds on a more encompassing view of what "discipline" is.

If Ada is the characteristic language of the "Closed World" (Edwards, 1996), Python in contrast represents the "Open World". It is inscribed with the culture of open-ended discussion and self-reflective practices of conventionalization, which are characteristic of the FLOSS world. In this cultural horizon, the classical analyses of hermeneutic philosophers are shown to be not adequate. To understand the enthusiasm of Python developers, we have to turn toward a philosophy that raises similar concerns, but argues for a positive appropriation of technology. Python programmers follow Robert Pirsig and his influential book *Zen and the Art of Motorcycle Maintenance* (1984) in conceiving

their own activity as one that is directed at artistic perfection, and provides shared enjoyment of co-operative work. Pirsig's philosophy departs from the classical critiques of technology in that it proposes a "Zen" way of life to "overcome the nihilism" inherent in the western rationalistic tradition of thinking, which is embodied in all the artifacts of our technological lifeworld. He emphasizes the importance of *craftsmanship* in relating to technical artifacts, which might appear as a romantic flight from the realities of modern processes of production, but some analysts, like McBreen (2002) have argued for its economic rationality in IT, and it is valued in the Python community as well.

In contrast with the classical hermeneutic reflections on the standardizing role of technical language, the practices of understanding of these Python programmers are better viewed as processes of *self-coordination*, standing close to the classical ideal of democratic scientific discourse and criticism, as Polányi depicts it in his utopian "Republic of Science" (1962). It is not a standardized horizon of understanding, maintained by a technologically embodied field of disciplinary power (as in the case of Ada), but rather the conscious *self-discipline* of free subjects that leads to the conventions and standards, which are indispensable for dealing with the ever-growing complexity of socio-technical systems.

Before we go into the details of our case studies, let's see the classical standpoints in the hermeneutic tradition about the status of technical language.

4.3. Technical language in the hermeneutic tradition

In 1957, at just about the same time as the Dartmouth Conference on the future prospects of Artificial Intelligence, and as the design of the first high-level programming language (LISP) took shape in the American military-industrial complex (Edwards, 1996: 257), Heidegger contemplates prophetically the "language machine" as something that will deeply influence the structure of human experience:

> The language machine regulates and adjusts in advance the mode of our possible usage of language through mechanical energies and functions. The language machine is - and above all, is still becoming - one way in which modern technology controls the mode and the world of language as such. Meanwhile, the impress is still maintained that man is the master of the language machine. But the truth of the matter might well be that the language machine takes language into its management and thus masters the essence of the human being. (Heidegger, 1957; quoted also in Heim, 1993: 8)

This perplexing vision – being as essentialist and romantic as it might be – seems also very disturbing in its plausibility (Dreyfus, 1998). If we reconstruct this argument based on the wider context of his work, it can be summarized as the following: Since language permeates our practices and our under-

standing of the world and ourselves, and also given that modern technology employs a range of controlled languages or *codes*, modern technology greatly influences our practices and our understanding of the world and ourselves. Technology intervenes "through mechanical energies and functions" in the lifeworld situations of language use, or more generally – to borrow a term from Lucy Suchman – in the contexts of *situated action*. How does this intervention take place? Since the field of our possibilities of perception and action are largely determined by the technological environment (Ihde, 1999; 2003), whenever we carry out actions via a command interface, solve problems through programming, or express ourselves in a markup language, the field of our possible interactions is preformed and restricted by controlled *technical codes* (Feenberg, 2000). Technical codes imprint specific techniques and patterns of practice on the situations of the technology's subsequent use, thus they acquire a character of *power* (as we are going to discuss it later).

To point at a similar example, according to the critique of Habermas (1987), the pathology of modern age is that the "system" – here he thinks of economy, power and (presumably10) technology as the *controlling media* of certain non-discursive forms of rationality – intrudes into the realm of "lifeworld", characterized by a discursive, hermeneutic form of understanding. This process of "colonization" withdraws moral, political and aesthetic questions from the realm of discursive understanding and subjugates them under the non-discursive rationality of economic, power and technological processes. In contrast with the discursive-hermeneutic communicative action embedded in culture, *controlling media* force communication into their reduced technical code, shaped by their narrow form of rationality. They reduce communication to the role of coordination of human action.

György Márkus states in a classical paper (Márkus, 1987) that natural scientists – and his argument applies to technologists as well – don't do hermeneutics, and they don't seem to be lacking it. His explanation is that the paradigmatic and specialized nature of research and the standardized scientific education grants them a shared background, which makes hermeneutics unnecessary in understanding scientific publication.

To sum up, Heidegger, Habermas and Márkus are concerned with the possibility that modern technology makes hermeneutic understanding dispensable, and replaces it with reduced – and thereby more efficient – forms of communication. They depict the domain of hermeneutic understanding as *distinct from* and *threatened by* the realm of modern technology. Thereby, they posit a schism between the two domains. If they are right, then the hermeneutic practices we have taken into the focus of our analysis are simply unwanted frictions in the technological machinery of software development, temporary problems that are going to be eliminated with the progress of technical systems.

On the other hand, while the severity of the impact of info-communication technologies on our lifeworld is undeniable, these philosophers are overshadowed by their generalized pessimistic and

---

10 For a detailed critical reconstruction see (Feenberg, 1996)

deterministic overtones, which are not widely shared by philosophers of technology anymore. Up to this day, technical codes did not coalesce into a unified mega-framework of thinking, not even in the realm of natural science or artificial intelligence research. Beside their standardizing tendencies, 21st-century communication technologies seem to stimulate various forms of democratic, open-ended discourse, creative self-expression and the free flow of information as well. Yet still, in their outmoded, essentialist fashion, these classics all address a very profound question, one that is still relevant up to this day: *how do people understand each other in the era of technologically structured and mediated interaction*?

Instead of viewing technology as a realm that stands separate from or intrudes into the domain of hermeneutics, I argue for a *continuity* between the traditional problems of hermeneutics – understanding different cultures, ancient texts, works of art, and ourselves – and contemporary practices of software development.

I'm siding with Andrew Feenberg (1996; 2000), Don Ihde (1990; 1999), Hubert Dreyfus (1997; 1998), Claudio Ciborra (1998), Lucas Introna (2006) and many other theorists in arguing that there is a need for an *empirical*, hermeneutic-phenomenological analysis of technology, and particularly ICT. I understand this analysis to be "hermeneutic" on two levels (Heelan, 1989; 1997). On the first level, we should recognize the importance of the activities in which technologically situated actors engage themselves to understand each other: the importance of hermeneutic practices *within* the realm of technology. On the second level, *our method itself* is a hermeneutic one: we reconstruct the meaning of the technical artifacts and rationales while re-contextualizing them in the cultural horizon, in which they were originally meaningful.

In contemporary philosophical literature, there are many other examples of positive hermeneutic appropriation of technical objects, and particularly, information technology. Robert Pirsig's enthusiasm for the art of technology is paralleled by Douglas Hofstadter's influential book about the interwoven threads of art, mathematics, computing, and philosophy, which illustrates the inherently paradoxical and open-ended nature of rationality even within these seemingly rigid frameworks of thinking (Hofstadter, 1979). Andrew Feenberg (1996; 2000) emphasizes the importance of the fundamentally cultural aspects of the appropriation of technology, which he calls "secondary instrumentalizations". Claudio Ciborra (1998) uses the late Heidegger's concept of the *Enframing* [Gestell] to interpret the users' lifeworld in various information infrastructure projects. Dreyfus and Spinoza (1997) reinterpret Heidegger's rich phenomenological rendering of *the thing* [das Ding] (Heidegger 1950) as having *both* optimistic and pessimistic consequences with respect to modern technology, and Bruno Latour (2004) – while condemning Heidegger's romanticism and sweeping critique – puts the techno-scientific *thing* right into the focus of his critical inquiry.

Now let's turn to a more in-depth study of the two contrasting programming cultures!

4.4. Concerns behind the design of the Ada programming language

The design of the Ada programming language is particularly interesting because it was governed by explicit intentions to impose certain practices on the users of the programming language. The design process was situated in an intensive discourse in quest for the "best" language, predominantly in terms of programmer productivity, reliability and (cost)efficiency. The series of Ada requirement specifications (Woodenman, Tinman and Steelman), the *Ada 83 Rationale (RATL)* and the *Ada Quality and Style: Guidelines for Professional Programmers* (AQS) are rich sources of reflections on the practices of the day, and they make clear the rationales behind the design decisions that left their mark on the language.

The general motivation was that in the early seventies, the US. Department of Defense (DoD) software projects saw an impending crisis of software reliability, and a Babelian confusion of software languages among the various development fields. The crisis was often attributed to the lack of expressivity of the languages used in design and development, and the difficulty of reusing proved solutions in new systems. These factors contributed to the disproportionate growth of software development costs. In 1973, Col. Whitaker started the DoD "Software Initiative", aimed to reduce the "High Cost of Software" (Whitaker, 1993; Ichbiah, 1984; daCosta, 1984). This was intended to reduce development and maintenance costs mainly by consolidating all DoD development under a unified language. However, the committee went much further than that: they were quite consciously designing a *community of praxis*, a *culture of understanding* instead of a computer language. The language features were selected to promote *coding practices* that were deemed beneficial: clarity, high abstraction, explicitness, code reuse and transferability of skills. The following excerpts show some of the main concerns of this standardization effort:

> Clarity and readability of programs should be the primary criteria for selecting a syntax. Each of the above points can contribute to program clarity. The use of free format, mnemonic identifiers and conventional forms allows the programmer to use notations which have their familiar meanings, to put down his ideas and intentions in order and form that humans think about them, and to transfer skill she already has to the solution of the problem at hand. A simple uniform language reduces the number of cases which must be dealt with by anyone using the language. (WOODENMAN - Needed Characteristics, reproduced by Whitaker, 1993)

> Readability is clearly more important to the DoD than writability. The program is written once, but may have to be read dozens of times over a period of years for verification, modification, etc. This is certainly true all weapons

> systems applications and even most of our scientific and simulation programs are very long lived. (TINMAN - General Goals, reproduced by Whitaker, 1993)
>
> Safety from errors is enhanced by redundant specifications, by including not only what the program is to do, but what are the author's intentions, and under what assumptions. If everything is made explicit in programs with the language providing few defaults and implicit data conversions, then translator can automatically detect not only syntax errors but a wide variety of semantic and logic errors. (WOODENMAN - Conflicts in Criteria, reproduced by Whitaker, 1993)
>
> The user should not be able to modify the source language syntax. [...] Changing the grammar [...] undermines the basic understanding of the language itself, changes the mode of expression, and removes the commonalities which obtain between various specializations of the language. Growth of a language through definition of new data and operations and the introduction of new words and symbols to identify them is desirable but there should be no provision for changing the structure of the language. (WOODENMAN - Needed Characteristics, reproduced by Whitaker, 1993)

With these criteria, the language designers address and try to avoid certain situations, where understanding among programmers traditionally breaks down. For example, maintaining and fixing a program code that was written by someone else is quite a bit of interpretative effort, because it involves the reconstruction of the original understanding of the problem situation from partially articulated traces. The criterion of explicitness addresses this. There is also the painful chore of deciphering existing solutions in order to adapt them to new situations. This is traditionally necessary because of the lack of "commonalities". In order to make a code "reusable" in future problem situations, the basic structures and conventions have to be standardized and the code has to be divided into independent functional modules. In Ada, the "package" language structure is designed to help this (among other techniques). A package hides the details of its internal implementation, and provides a well-defined set of functions. The user of the package can use it as a "black box", without needing to know the internals. She can go on thinking of her problem at the "higher" abstraction level of functional modules, instead of bogging herself down with the details. "Higher" refers here to the order of closeness to the details of the machine operation. In the order of closeness to the conceptualization of the problem, it is actually a *lower* level of abstraction (Smith, 1987).

The specification states explicitly that the programming language serves not only as an interface with the computer, but as the linguistic medium of the programmer community as well, in which they articulate their problems, intentions, assumptions, skills and ideas. The occasional misunderstandings between project members that arise in this communication through shared source code often results in reliability failures. Implicit assumptions, unexpected results from another module are the major culprits. The designers are trying to overcome this with explicitness, redundant definitions and increased readability of the control structures. I must remark here that the belief – apparently held by the designers – that there exists a totally transparent and explicit formulation of any problem reveals a certain epistemological naiveté of the language committee. This has been contested by many philosophical theories of knowledge, because when we start to explicate thoroughly our background knowledge, we're starting to build implicitly on an even broader set of background knowledge, again in need of explication (Winograd and Flores, 1987). Suchman even argues that in the case of *plans,* – and the argument applies to programs as well – it is their inherent implicit *vagueness,* which makes them usable under varying circumstances.

Significantly, what is common in all the episodes of misunderstandings is that they all involve open-ended hermeneutic efforts with unpredictable outcomes. These hermeneutic episodes are pictured by the language designers as unwanted, because they introduce unpredictable delays and further errors, and thus constitute a project risk. Unreadable, hard-to-decipher, "spaghetti" code on the other hand puts the original developer in a privileged position, because he is the only one who understands it: it can easily make her the irreplaceable "key figure" of the project (Boehm, 1979).

But the elimination of hermeneutic practices for frictionless communication is not in the interest of the wage laborer at the lowest level of corporate hierarchy. DoD specialists often "stress the low skills and motivation of most military programmers" (Kling and Scacchi, 1979: 34). Short-term deadline pressure overrides subtle concerns. Source code beauty also ranks very low on the programmer's priorities list if she doesn't have any feedback on the long-term costs of her non-understandable code (Kling and Scacchi, 1979: 37). She might even be proud of her ability to solve hard hermeneutic problems with her unique skills, and might also get rewarded with wage bonuses for doing that (Boehm, 1979). And furthermore, being irreplaceable by having exclusive understanding about an exotic system means secure employment for her. Her "enlightened self-interest" lies in the complexity of the code. How could she be motivated to think abstractly and modularly, to write code not for herself, but for her successors, who are going to reuse it (Smith, 1987)? Or more generally: how can her interests be reconciled with those of the management (Kling and Scacchi, 1979)?

In order to answer this question, we have to see that the Ada initiative is an attempt to transform thoroughly the way in which programming problems are perceived and articulated by the programmer. The concerns of the higher management are carefully designed right into the structure of the language. From now on, the programmer can't even conceptualize her problem without considering these concerns, since they are already inscribed in the use-patterns of her conceptual tools. Even if

the Ada programmer were not consciously aware of these concerns, she would have to conform to them. She couldn't help but share her understanding with her workmates. With Ada, her entire motivational structure changed. This transformation took place at the level of language skills and in the patterns of interaction.

What makes the programmer follow the strict rules of the language? How is this kind of *discipline* to be established? The „regulation" of the „possible usage of language" is achieved by means of a strictly specified, materially embodied *compiler* program. It does not only constitute the interface between the programmer and the machine, it also has a *disciplinary function*: it simply doesn't let such code through, which doesn't conform to the intentions of the language designer. *The compiler is thus a political artifact, working as an obligatory passage point* (Lessig, 1999; Latour, 1992).

In order to protect the technical code of the compiler with the social code of legislation (Lessig, 1999), the designers were even so cautious as to register "Ada" as a trademark, in order to be able to revoke the right of using the name from compilers that do not fulfill the Ada specifications. This was necessary because the DoD wanted to make sure that its subcontractors – with the collaboration of compiler companies – won't use a simplified, relaxed implementation of Ada.

The *type* of all variables has to be defined in every function definition, at the beginning of every code block; otherwise the compiler stops with an error. This so-called *strong* and *static* type information is often wholly redundant. Beside performance reasons, it serves chiefly as a kind of "double-entry bookkeeping", a redundant security check that forces the programmer to keep account of, for example, the kinds of variables that can be used as parameters for a certain function.

These rationales behind the language design can be understood from the perspective of management. The first paradox of software project management is that it has to persuade people to act according to abstract principles and long-term interests of the management, even if they might not fully grasp them or even if their short-term personal interests directly conflict with them. They also have to do this creatively and at a high abstraction level. It is far from obvious, how one can make people to do that (Gerhardt, 1989; Kling and Scacchi, 1979). For example, programmers and middle management normally might admit that type information helps to avoid reliability failures on the long run, but if the language didn't force them, the benefits were so abstract and indirect, that they would often not bother themselves to supplement the code with such documentation consistently. Back then, in the age of printer-terminals and paper-based documentation, it would have been indeed hard to do so (Brooks, 1995). With Ada, the situation was to be changed. The compiler was to work as an abstract "electric fence" as it forces the programmer to think the *right* way.

The second paradox of software project management is that it has to enforce understanding among the various actors taking part in the project, often rooted in different lifeworlds, coming from different backgrounds of understanding. Having different horizons „removes the commonalities" and that runs the risk of a breakdown of understanding. In such a case, one has to engage in an intensified hermeneutic effort, which is perceived by the management as a factor of risk. So the question is:

How can the management ensure that the (partial) horizon-sharing takes place *predictably*? To make the question more specific: how could two developers be forced to share each other's perspective of the system, so that they won't misinterpret the data coming from the other's code? How are they supposed to be forced *not* to build false expectations about the output of the other's code?
As a solution, the structure of Ada reflects the development hierarchy at an even more specific level. The organization of functional modules (packages) in the produced artifact is supposed to mirror the hierarchical organization of work by delineating self-enclosed units that can be managed intellectually by a responsible individual or a team (Ichbiah, 1984: 994). The pathways of communication and control transfer between the modules – and thus between associated developers – have to be declared explicitly in the beginning of the source code of each package, together with the type information of the transmitted data. Later versions of Ada have a dedicated language feature called *interface declaration*, which brings all communication between software modules under the control of the compiler, to enforce obedience to the requirements stated in the interface declaration. Interface declaration and programming can even be separated: the compiler guarantees that a handful of interface designers can enforce mutual understanding between many implementation programmers – according to the long-term interest of the project. This induces a certain social stratification, a power-hierarchy between large-scale designers and small-scale developers. It is ironic that the proverb "Divide and conquer", used often by programmers to refer to modularization (e.g. DeRemer and Kron, 1975), is at the same time a management strategy played upon them!
If we look at the definition of power given by the late Foucault, we can see that it is highly relevant in this case:

> [T]he exercise of power [is] a way in which certain actions may structure the field of other possible actions. (Foucault, 1982)

The programming language – as a product of the actions of its designers and its implementers – is, in this sense, a field of power, because it structures the possibilities of action, and thus the field of hermeneutic practices in which its users can take part.
The programmer perceives her problems and carries out her actions within this field of possible actions, and her perspective is already aligned with that of the management. However, for the programmer, this built-in perspective rarely ever gets into the focus of thematic understanding. She keeps her minute problems and tasks in her mind, and engages in a code-compile-test cycle, while trying to avoid compiler errors. The agent exercising power over her is not personally present; sometimes it is not even identifiable as a particular individual or a group. However, it is there, and it guides the hand of the programmer while writing code because it has been inscribed into the biased design of her tools and her technical lifeworld. Finally, just like in the case of Bentham's Panopticon, the power field gets internalized by the programmer in the form of routines, skills and conceptual

frameworks, by which she orients herself and copes within the technological lifeworld. (In this aspect, ours is a bit more encompassing definition than those of contemporary reflections of the relation between computing and power, focused on explicit threats and rationales, see (Kling, 1974))
The designers of the programming language thus implement a modernistic tendency: they draw a line from the management perspective between what they consider "normal" and "deviant" programming practice and then they intervene into the structure of technologically mediated practices to bring the behavior of the programmer under control.
The experiential aspect of this situation can be characterized with Heidegger's notion of *the They* [*das Man*]. The They is the mode of our existence in which our perception and action follows modes that are determined by others:

> We enjoy ourselves and have fun the way *they* enjoy themselves. We read, see and judge literature and art the way *they* see and judge. […] [T]he they maintains itself factically in the averageness of what is proper, what is allowed, and what is not. Of what is granted success and what is not. This averageness, which prescribes what can and may be ventured, watches over every exception which thrusts itself to the fore. (SZ 127 / BT 119)

When we act in the mode of the They – and this is the typical mode of everyday action –, we are following intentions that are not ours, but are so deeply engraved in our practices that we cannot even articulate them, or imagine doing otherwise. This is the case, for example, when programmers forget about the original rationales built into the language, and rationalize source code aesthetics (falsely) by referring to mathematical principles or folk heuristics (like "elegant solutions are always more efficient").
The mode of existence of the They – according to Heidegger – is characterized by *averageness* [Durchschnittlichkeit] and *dependency* [Unselbständigkeit]. We often follow normalized practices, and we depend from those – including ourselves – who shape these practices. The They also *disburdens* [entlastet] us from the burden and responsibility of many important decisions, because these decisions are already built into the normalized practices themselves, which we follow unquestioningly.

> However, because the they presents every judgment and decision as its own, it takes the responsibility of Dasein away from it. [...] It can most easily be responsible for anything, since no one has to vouch for anything. (SZ 127 / BT 119)

As the Ada standards explicitly state, the user is not allowed to make changes to the linguistic framework. Individual initiative would break down commonalities and pose risks, so it is discouraged. If

the language designers' intentions were taken to their logical consequences, then the programmer won't have to learn theories about developing reliable software, or about frictionless communication. It would suffice to obey the syntactical and stylistic rules of the language. She also won't face dilemmas between short-term and long-term goals, between management's and own interests, because others will have already taken care of these decisions. This is the essence of military hierarchy, embodied in the language – this *disburdening* is what makes modern wars possible in the first place! At the end of this process, when no special ability is needed anymore for the successful, frictionless development, except for the knowledge of the language, the place of the individual programmer can be filled by *anyone*. The once-admired hacker will give place for the average, normalized, replaceable cogwheel of the development machinery.

Of course, back in the seventies, this reflected well the needs of the DoD projects, which involved many subcontractors, employed hundreds of programmers and encountered high fluctuation over their very long time spans. The *averageness* and *dependency* seemed to be just the necessary conditions to build efficiently such highly complex technological systems as the F-16 jet fighter (Whitaker, 1993) – and they are still relevant in most contemporary software development organizations.

4.5. Conclusions drawn from the Ada case

At this point, we might be expected to conclude that Habermas and Heidegger are right in their pessimistic visions: hermeneutic practices are indeed being replaced by a simplistic technical code, which serves solely the purpose of the coordination of human action. Nevertheless, there are severe problems with this conclusion. First, it is not a reified „language machine" or "Technology" with a big "T" what "masters the essence of the human being", but it is rather a collective act of managers, programmers and political decision-makers acting under specific historical conditions. People in the management of the DoD had good reasons to mirror a military hierarchy in the language: they wanted to win the Cold War with their limited resources. In their cost-saving effort, they (as *the They*) represented the American taxpayer. Their goals and means do not come from a trans-historical reality, but emerge from a wider-scale, historically situated political discourse. When this discourse takes a different turn, when the specific historical conditions change, the process can take a wholly different trajectory, as the subsequent history of Ada illustrates. When in 1987 political decision-makers mandated the exclusive use of Ada in all DoD projects through DoD directive 3405.1, it created a protected space within which a large development culture started to flourish. The software of the F-16 jet fighter and the Boeing-777 airliner are the greatest results of this era. This means that Ada was technically successful, as quantified studies have also shown (Reifer, 1987, 1996; Whitaker, 1993). But soon after when in 1997 Emmett Paige, the Assistant Secretary of Defense lifted the requirement for DoD projects to use Ada (Paige, 1997), the market share of the language went into steep decline. This decision reflected a change in strategy from the part of the DoD: they took a different approach to avoid the problems that were originally addressed with Ada. *Instead of focusing on the language, they*

started to focus on regulating the general patterns of the software engineering process, with all its communication and inscription-producing practices (CPPCUADoD, 1997). Ada barely survived in the commercial world, even after its success in the protected market of the DoD. In 2006, it made a headline in the AdaCore newsletter that Boeing chose Ada for the control of the *air-conditioning system* of the Boeing 787[11].

This shows that the technical code that is aimed to displace hermeneutic practices is still subject to societal discourses at the meta-level. As we're going to see acutely in the case of Python, explicit normalization at the level of the language is only one approach among many. There are many alternative ways to stabilize development processes, and these are subject to diverging measures of success within the social context.

Nowadays Ada contributes much less than 1% to the software developed worldwide, even on the embedded platforms for which it was designed. There is also a wide proliferation of various other programming languages, like Python, which are founded on principles that are contrary to those of Ada. The „mega-machine" of the DoD also gave place to many new forms of organization in software development, like eXtreme Programming (Beck, 1999) or the Agile movement (Hunt, 1999), due to various changes in the social world, copyright laws, etc. (CPPCUADoD, 1997; Feinberg, 1987).

Furthermore, Ada cannot be unanimously taken to be a success even according to its own aims. It was widely acclaimed to be hard to learn, hard to use, and its strict syntax prohibited the use of certain abstractions generally considered handy (e.g. conditional compiling). Beside all its strict rules, it still had to be supplemented with a 193-page long style manual (AQS), just like most other programming languages. The productivity increase generally attributed to Ada, measured in function points and number of lines of source code written per day (Reifer, 1987, 1996), might as well be attributed to the verbosity of the language, instead of its ease of use. More detailed criticism can be found in (Baker, 1997; Bennett et al., 1982; Dijkstra EDW658-663; Feinberg, 1987), which argue that the "strong" typing system is in fact too weak in many important situations, whereas in others it entails unnecessary bureaucracy.

Disburdening language users by taking away from them the power of making local decisions about their own practices can have negative effects as well. "Many social interactions [...] have a »local rationality« which may not [be] visible in (assumed) global perceptions of common computing environments.", argue Kling and Scacchi (1979: 39), or, in other words, the use-contexts envisioned by the designers might be at odds with reality (Kling and Scacchi, 1979: 30). The criterion that the core

---

11 http://www.adacore.com/2006/05/01/hamilton-sundstrand-selects-gnat-pro-for-boeing-787-air-conditioning-pack-control-unit/

language cannot be extended kills any individual initiative from the part of the compiler or tool developers.

Finally, as some analysts point out: "When higher quality, lower cost expectations were not immediately realized, Ada was blamed." (Kerner, 1992; CPPCUADoD, 1996). Frustrated with the language and the practice they did not choose, users and decision-makers often shifted the responsibility for their mistakes to the absent language designers.

So far, the theories of Heidegger, Habermas and Márkus seem to be very consistent with the design rationales built into Ada. What they fear is what the language designers explicitly aim to achieve. The problem is that the validity of these rationales has proven at least questionable by the concrete history of Ada. Now let's turn to our next case study, which stands at the other extreme of the programming language spectrum with respect to hermeneutic concerns.

4.6. Open-source languages: the case of Python

Having seen the sophisticated design rationales behind the language of the military, the following question might spring into the reader's mind. If it took such a sophisticated design to avoid communication breakdowns and ensure shared understanding in the case of Ada, how come that individualistic hobbyists in the FLOSS world can develop complex, high-quality software systems without similar, highly centralized, hierarchical bureaucracies, supported by the language? Particularly, how can it be that FLOSS source code does not tend to degenerate into incomprehensible, "spaghetti" code even in the case of C and Python, both of which contain language features that make them much more prone to this than Ada is?

I'd like to demonstrate that concerns about understanding each other's code are just as important in the FLOSS community as was in the DoD, but here, as opposed to the DoD, they take a more hermeneutic approach to achieve that.

4.7. Constructing „pythonicity": the Zen of Python

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.
Namespaces are one honking great idea -- let's do more of those!

(Tim Peters, "The Python Way" on python-list, 04.06.1996, also PEP 20)

This is the "Zen of Python", a summary of the values esteemed by the Python developer community. It is so standard that there is even a built-in statement in the interpreter, which prints out this list: import this. Some take the list to be a joke, but these rules are often referred to in various arguments about design decisions (e.g. between 1999 and 2007, "Explicit is better than implicit" is mentioned individually on the python-dev list 79 times, and on the python-list 303 times[12]), and they were introduced explicitly with that aim in mind (Tim Peters, "The Python Way" on python-list, 04.06.1996). This is obviously not an exhaustive list of logically independent "founding principles": it is better to conceive them as heuristics that are somehow characteristic of the *general framing* of arguments used in debates. A solution is called "pythonic" if it is in accord with these rules. Of course, what to call "pythonic" is not easy to settle upon: the rules are inconsistent not only with some other standard practices (as Ian Bicking points out in the "UnZen of Unpython", there is at least one good reason to argue against each rule), but they also conflict with each other. (For example, "flat is better than nested" seem to stand opposed to the hierarchical namespaces – comparable to Ada packages –

12 Notes on the data collection methodology. The archives of python-dev and python-list mailing lists were downloaded at 03.10.2007, with total number of postings 70619 and 375050, respectively. „*Individual mentioning*" refers to the *number of postings*, where the searched keyword appears in the *non-cited parts of the body of the message*. A relatively simple Python script was used for gathering the statistics, using line-oriented regular expression matching, which does take into account the cases where keywords are used with syntactic variations or span between lines. Partial word patterns ("extensib.*") were used to cover a minimal degree of syntactic variation (extensible, extensibility, unextensible, ...). The statistical data on the comp.lang.python, .perl, .ada and .javascript newsgroups were gathered with the search facility of groups.google.com, which does not allow for differentiation between cited and non-cited body text. Syntactic variations of the keywords were introduced manually, and the results were aggregated. The numbers refer to the number of postings, in which the searched keyword appears. The difference between the data collection methodologies does not affect the validity of the argument for which it is introduced as supporting data.

praised in the last line.) In order to use them as arguments, they have to be interpreted and argued in each concrete situation, and the last word is always that of Guido van Rossum, the original designer of the language.

The endeavor to relive technology within a "Zen" way of life, to rejoice in the artistic moments of engaging technology has its ideological roots in Robert Pirsig's influential book *Zen and the Art of Motorcycle Maintenance* (1984). Pirsig argues there that the nature of "Quality" defies definition and explicitness and it can't be codified in a style manual, because it must be conceived as a general orientation toward artifacts, people and ourselves. Every fragmented articulation of "Quality" can only serve the purpose of transforming the orientation of people, instead of exhausting its meaning. The effort to rationalize quality turns it into an external set of rules, a form of disciplinary power which instills a rule-following "slave mentality" (Pirsig, 1984: 199), whereas real quality stems from the creative and responsible interpretation of the lifeworld situation by free and self-motivated people. We are going to assess, to what extent is this ideological background reflected in the actual practice of developers, and at what price comes this "freedom".

Talking of hermeneutics, the most relevant feature of this "zen verse" is that *at least 10 of the 19 rules argue for the easy understanding of source code* – just as the Ada specifications do! It is a plea for a modernist aesthetic of simplicity, practicality and order, but praises it only to the extent to which it helps to make the code easier to understand (Rossum, 1996). And understanding is indeed very important in the case of Python. Since the typing system is not static – like in Ada – but dynamic, it is not easy to tell, for example, what kind of parameters can be used to call a function, and what kind of result will it give back. Since there are no type-checks neither at compile-time, nor when the parameter is passed to the function, bugs only appear when an assumption about the parameter breaks deep within the function (Alex Martelli, "Inheriting the @ sign from Ruby" on python-list, 12.12.2000). It is thus essential to make the assumptions explicit about the function's parameters – for example, by using meaningful parameter names, by providing relevant comments, or by communicating it directly to the colleagues. There is a built-in language feature called "docstring" which makes documentation comments easy to look up and use; in most other languages, this requires extra tools. There is no built-in language feature to enforce commenting, though. The "double-entry bookkeeping" mechanism of Ada (and the associated burden) is missing. If one omits to make crucial information explicit for her fellows through clear source code or through clear comments, it will entail no immediate sanctions. This means that in principle, it is quite easy to write incomprehensible programs. It can be made very hard to guess from the source code what goes on at runtime.

However, programmers choose Python because for them, dynamic typing entails a kind of "freedom" or "flexibility" (DH, "Python vs. Lisp -- please explain" on python-list, 19.06.2006). One doesn't have to build up and define a type hierarchy before she calls a function with a certain object: she just writes the call. It's up to her to ensure that the object meets the anticipations in the function.

The burden of the redundant administration, associated with explicit typing is lost. This makes Python programs simpler, more compact, and easier to experiment with.

In general, the designs of the two languages endorse *different patterns of use*. Ada assumes that interfaces will be defined beforehand by an elite designer group, and then the lower-level programmers will proceed with the implementation. Python is designed towards interactivity and rapid application development within small developer groups. If there is an error, you can start a console and try to explore the possible causes by manually invoking the functions with different arguments, without going through the process of redefining the interfaces, adjusting the code and then recompiling the project. In the case of Ada, experimenting with the parameters involves finding and editing relevant code fragments distributed throughout the code, because only such a project can be compiled, which is consistent in all its type definitions. In the case of Python, however, you can easily experiment with the code, but the compiler won't warn you if you leave the code in an inconsistent state. Bugs will only appear at runtime, during testing or even real-life use – a luxury that the designers of mission-critical applications in the DoD just couldn't afford.

The only way to ensure that a Python project is in a consistent, functional state is through extended testing, generally done by employing automatized test suites. Python uses a structured error-handling mechanism somewhat resembling Ada's, which makes testing and debugging easier. Nevertheless, you would also need a test suite in all other languages, since even the best compiler catches only a fraction of the bugs. As the Ariane 5 Flight 501 Failure report (Lions, 1996) also reveals, even a syntactically correct Ada program can encompass false assumptions about the parameters. The existence of language facilities geared at increased reliability does not warrant reliable systems: their deployment has to be designed and reviewed carefully (for influential contemporary reflections, see also Borning, 1987; Parnas, 1985). Python shifts most of this responsibility from the language to the developer and his testing group.

At this point, it might seem as though Python totally lacked strictness. This is not true. Python is strict where it is considered necessary by the community, but that often does not coincide with previous norms of strictness. The most obvious example is the syntactic rule of indentation. Every embedded code block must be marked with an indent of four spaces or a tab character. In most other languages, code blocks are marked by some kind of parenthetical constructs ({ … }, begin … end). Indentation in these languages is not obligatory, although it is generally required by style standards.

It is almost ironic that either because Chomskyan theories of artificial languages were so closely modeled after natural languages, or because of the legacy of FORTRAN, it hadn't been considered previously that *the number of spaces, tabs, and line breaks* can have a *syntactic* purpose. Indentation not only improves readability enormously (Rossum, 1996), but it makes many parenthetical constructs – which tend to clutter up code – redundant. In most other languages, spaces, tabs and often line breaks as well are treated as undifferentiated "white space", which leads to infinite variations on coding style. (Rossum, 1996; Arnold, 2004)

Ada and Python are both designed for readability and understandability, but they have different conceptions on what they take to be "readable". *In the case of Ada, "readability" is explicitness and verbosity, while in the case of Python, "readability" is simplicity and terseness.* In the DoD community, "readability" is enforced by static typing and standardized coding style. You write understandable programs because you are disciplined by the compiler (and the legal code explicitly backing it). In the Python community, "readability" is also influenced by syntax, but as the "Zen of Python" succinctly reveals, even more emphasis is being laid on building the shared culture, in which one feels responsible and motivated to be helpful to her fellow programmer. The general impression is that you should get feedback from your fellows and your customers, but only rarely from the compiler (interpreter). In many cases, it is up to the various user communities' choice to settle upon standards and conventions:

> "A universal convention supplies all of maintainability, clarity, consistency, and a foundation for good programming habits too. What it doesn't do is insist that you follow it against your will. That's Python!"
>
> —Tim Peters on comp.lang.python, 2001-06-16

> If you violate these conventions, the worst you'll get is some dirty looks. But some software […] will be aware of the conventions, so following them will get you the best results.
> (Python Enhancement Proposal (PEP) 257: "Docstring Conventions")

> A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is most important.
>
> But most importantly: know when to be inconsistent -- sometimes the style guide just doesn't apply. When in doubt, use your best judgment. Look at other examples and decide what looks best. And don't hesitate to ask!
> (Python Enhancement Proposal (PEP) 8: "Style Guide for Python Code")

This approach, besides emphasizing global conventions, leaves ample space for decisions at the *local level* of development. The progress of language evolution, first through mailing-list discussions and reviews, and in recent times through the semi-formal process of Python Enhancement Proposals (PDP, PEP 1, PEP 42) also reflects this democratic spirit, and this results in swift changes of the core language (Rossum, 2001).

On the one hand, the "Zen of Python" and the PEPs form part of a *conservative* strategy: they are being employed by senior developers to fend off those, who want to "bend [the language] into uncomfortable positions" (Patrick Phalen, "The Python Way" on python-list, 03.06.1999). In other words, "Like a FAQ, which tries to reduce newsgroup traffic by answering questions before they're asked, PEPs try to reduce repeated suggestions." (PDP). They do so by explicating the rationales and the shared values that went into each design decision.
On the other hand, the "Zen of Python" is always open to reinterpretation, and PEPs are often revised, if there is enough community support to do so. That the democratic principles laid out in (PDP) are indeed in effect can be demonstrated by the fact that on average, 1.72% of the postings on python-dev are votes conforming to the Apache Project voting scheme (See fig. 1.)[13].
In contrast, such local overriding of global conventions was perceived to be the root cause of reliability and cost problems by chief Ada designer Jean Ichbiah (Ichbiah, 1984), because in his view they would inevitably lead to disintegration. Thus, even if the Ada standardization process was open to peer commentary (some 7000 comments were considered) (Ichbiah, 1984; Boehm, 1979; daCosta, 1984), it never resembled the openness of the PEPs. The basic requirements and assumptions remained fixed, and any further extensions were banned. Ichbiah insisted upon that

> [...] a design like this has to be done with a single strong leader, since it is very important that the major architectural lines of a language be kept consistent: Consistency can only be achieved with one person defining the major lines. (Ichbiah, 1984: 997)

Python also has a charismatic designer (mockingly called the "Benevolent Dictator for Life"), Guido van Rossum, who was solely responsible for final decisions on language design questions up until 2000 (Rossum, 1996, 2001). Although he is the one generally attributed for the conceptual integrity of the language, we have seen that the Python evolution is much more decentralized and flexible than Ada standardization.
As we can see on (Fig. 1.), words like "readab(-le, -ility)" and "convention(-s, -alization)" appear in Python-related mailing list/newsgroup postings with at least as, or even greater frequency than in those dedicated to other languages. The frequency of postings mentioning specifically Python-related understandability issues, such as "implicit(-ness)", "explicit(-ness)", and "indent(-ation, -s, -ing, ...)", is significantly higher than in other forums. On (Fig. 2.) it can also be seen that these ratios are resulting from a sustained interest, instead of an already settled debate. It is also worth pointing out that the frequencies of the postings using these words are correlated.

---

13 Counted as individual mentioning.

These findings are *signs of an ongoing process of reinterpretation and renegotiation* of what is understandable and how to arrive at shared understanding. *This discourse is a characteristic example of what I call "hermeneutic practice".*

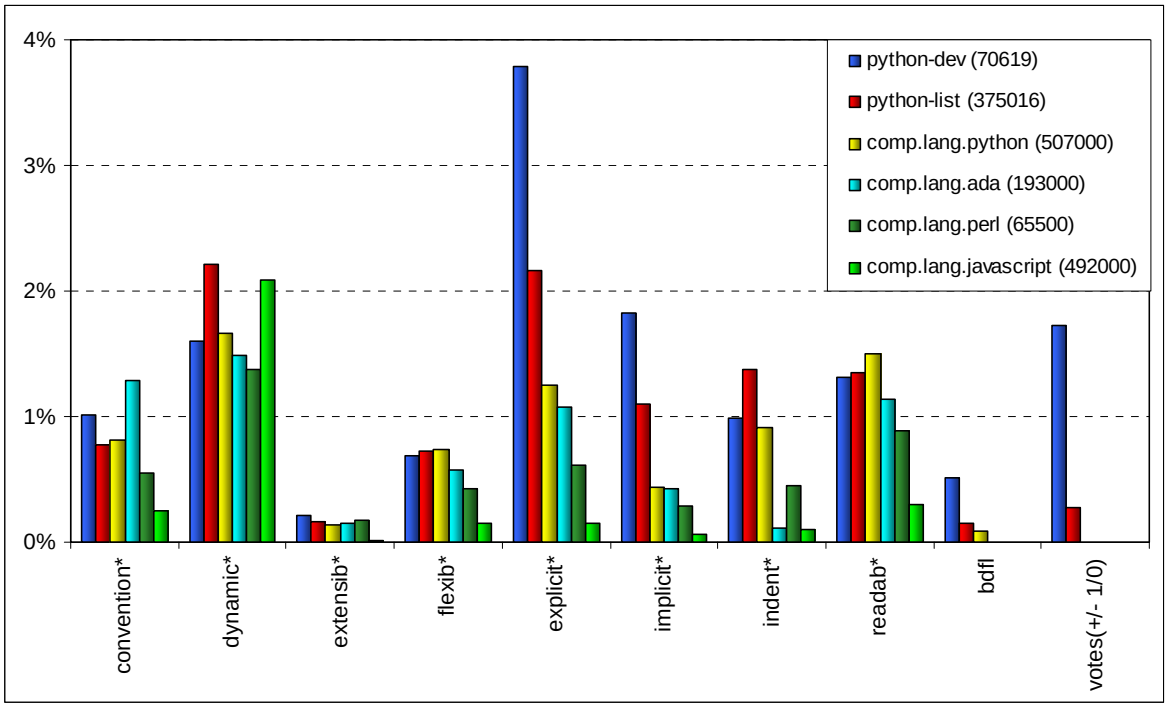

Fig. 1. Percentage of mailing list/newsgroup postings containing the respective keywords. (Figures for python-dev and python-list refer to occurrences in non-cited body text, whereas comp.lang.* newsgroup figures contain all occurrences. The higher frequencies of python-dev and python-list are to be interpreted with this bias in mind. The total number of postings is shown in parentheses.)

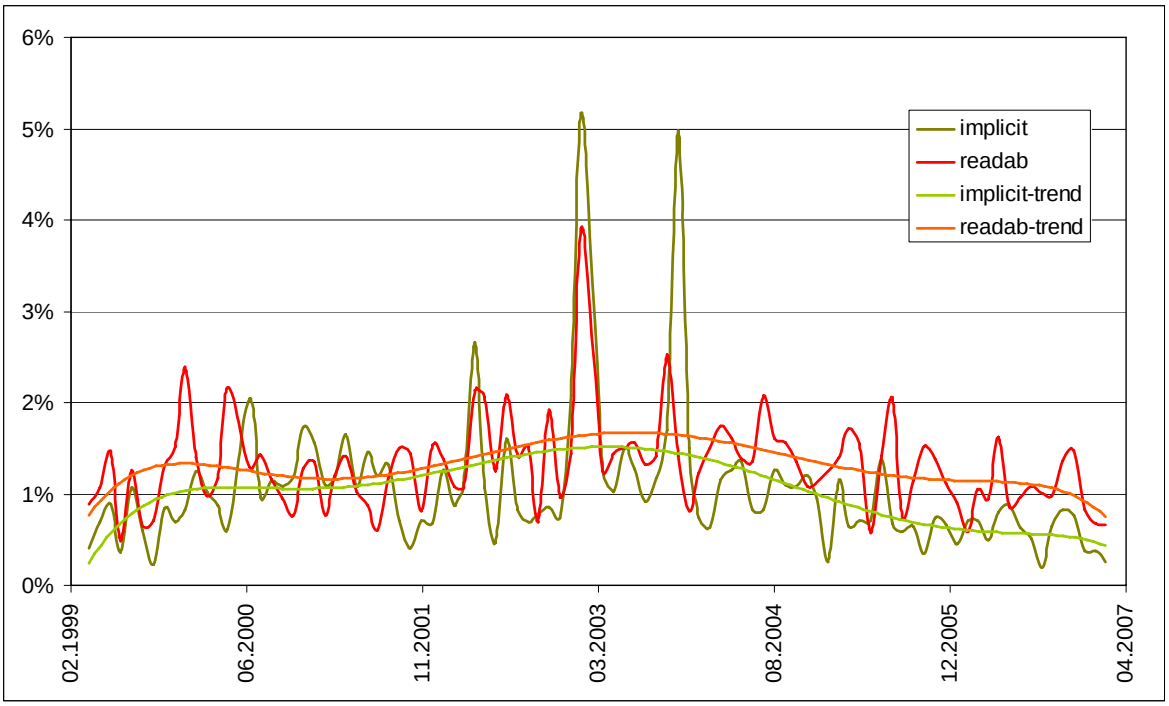

Fig. 2. Correlation between the relative frequency of pattern "implicit" and "readab*" in python-list postings. The correlation coefficient is 0.502.

4.8. Conclusions drawn from the Python case

We can turn now to answering our original questions. Ada and Python are two paradigmatic cases of technical codes, but Python is not a technical code in the sense it is feared by Heidegger and Habermas. We must admit that it does try to reduce the number of decision alternatives open to the programmer, as any well-designed artifact should do, according to leading design theorists (Norman, 1998). "There should be one – and preferably only one – obvious way to do it." – that means, there should be only one preferred language idiom for standard problems, so that people will tend to use in the same way (Rossum, 1996). But this is not strictly enforced (James J. Besemer, "Dijkstra on Python" on python-list, 13.08.2002), and contrary to Ada, it is not the invisible, biased field of power, unconsciously transforming the phenomenal world of its users, according to the intentions of absent language designers. Python does not try to restrict the user's field of interaction in order to force her to adhere to a specific standard of readability and a specific ideal of order: to a great extent, it is up to the user's community to take a stand on these matters.

Answering our other question, FLOSS software development doesn't disintegrate, because "free" software development does not mean "anarchical". Instead of having strict methodologies and controlled technical codes, their conventions emerge in rather highly structured and self-disciplined hermeneutic practices. Most contemporary FLOSS projects, – like the Python community of our

next case study – employ discussion groups, wikis, convention manuals, and various other means of communication, which decentralize decision-making and hermeneutic activities. Conventionalization is often based on *self-coordination,* instead of the decisions of an appointed elite expert committee.
Why does this approach seem to work so efficiently? Ada wants to standardize and automatize one of the highest levels of intellectual work. But as DeMarco and Lister conclude their arguments in their classic book "Peopleware":

> When you automate a previously all-human system, it becomes entirely deterministic. The new system is capable of making only those responses planned explicitly by its builders. So the self-healing quality [characteristic of human systems] is lost. […] If ever the system needs to be healed, that can only be done outside the context of its operation. […] If the […] system has a sufficient degree of natural ad-hocracy, it's a mistake to automate it. Determinism will be no asset then; the system will be in constant need of maintenance. (DeMarco and Lister, 1999: 113-114)

The Python language and the community itself, by having a "sufficient degree of natural ad-hocracy" or "flexibility", can accommodate itself very fast to new challenges and new business scenarios. Particular conventions of understandability are "factored out" of the core language, so they can be renegotiated in each user community, according to their particular needs and valuations. In certain user communities, conventions might gather a strongly conservative momentum, whereas in others, they might remain only loosely coordinated. Furthermore, beside that the core language is constantly being extended and reformed through Python Enhancement Proposals, it is in itself so flexible, that you can implement, for example, your own type checking mechanism (e.g. http://www.ilowe.net/software/typecheck/).
In Ada, you are forbidden to alter or extend the core language, given the fears from breakdowns of understanding and compatibility. This rigidity right in the center of the most flexibly changing commercial scene might explain the commercial failure of Ada (Kerner, 1992), but it can be generalized to the concept of "technical code" itself: that human systems working under overly rigid technical codes do not seem to have the necessary "self-healing" quality to survive in evolving markets. Python's flexibility might prove to be a factor, which makes it a *disruptive technology* in certain markets (Christensen, 1997).

4.9. Python in action: hermeneutic practices in a FLOSS project

The importance of hermeneutics in the case of Python is clearly focal. Lacking a rigid technical code, developers in a healthy open source project (like the one in the following example) often produce much more informal communication and documentation than source code, because they have to ne-

gotiate their standard practices and their conventional solutions, articulate their background assumptions, reinforce their shared values, and finally, to reassess their position in the greater cultural and economic context. The shared forums and discussion groups are buzzing with messages, developers and users are engaged in an open-ended, multimodal discourse. It is only in this context, in which the shared source code is intelligible.

In order to get a glimpse of the hermeneutic practices they're engaged in, I take a closer look at how they categorize types of communication and how they employ various *new media* channels for each category. I'll take as an example the buzzing Python web framework development project at www.turbogears.org.

The most important feature of their communication is its striking *multimodality*, made possible by the relative cheapness and general availability of diverse communication media. Right from the starting page, you can access screencasts, tutorials, blogs, discussion groups, announcement notifications, IRC channels, wikis, books, and a few clicks deeper you have the option to use the issue tracking interface (Trac), browse the source code, see recent changes, "to do" lists and more. These new media provide specific mechanisms for various categories of communication. Screencasts and tutorials are appropriate to initiate a new user in the project. Books and wikis contain relatively static conventions and background knowledge. Discussion groups and IRC channels are informal forums where one can discuss new ideas, ask questions when she's stuck, and discuss possibilities of resolving problems. Blogs are used for announcing new directions, sharing success stories, showing off with witty solutions, and – last, but not least – for the self-promotion of the participants. The Trac is an issue tracking system, which simplifies the chores of keeping track of bugs, feature requests, "to do"-s, milestones, relevant discussions, the responsibilities and the actions taken to resolve certain issues. But it is also a forum, in which alternative solutions get reviewed and debated. From the Trac, one can easily browse the history of modifications, the relevant parts of the source code, the comments about each modification, the discussion that preceded it, and the specific considerations that resulted in the particular solution. Whereas for Ada, the primary communication medium was source code itself, here almost every line of the source code gets interwoven in the multimodal discourse: it is a big mistake to think of the resulting code without taking this context into consideration.

How can it be that developers don't get bogged down with all these discussions going on, answering messages and so on? These discussions have a significant effect on the quality of the resulting code and the morale of the developers. Thinking through a solution collaboratively while documenting the process has its analogue in the academic process of "open peer commentary", and it is believed to actually *save time* for many generations of subsequent use: first, because it results in a superior solution, second, because it results in a wider consensus over new conventions, and third, because new users can search the discussion archives and see the considerations that went into the code, so they can take them into account (or even revise them) without generating further requests for clarification. Such requests will be answered with something like "Have you already tried Google?" or with a

link referring to the original discussion. Instead of trying to satisfy the *decontextualized* criteria built into the compiler in order to make the code readable, developers have to persuade their peers directly that their solution is understandable within the *situated* context of the system (Suchman, 1987). Here is a wiki snippet to show how this discourse is structured and how each communication category is confined to its specific medium:

> How to Submit a Patch
>
> 1. Post the issue to the mailing list (turbogears for bugs, turbogears-trunk for features)
> 2. Discuss it and determine it is really a bug/useful.
> 3. Make a ticket on Trac.
> 4. Attach a patch and put [PATCH] in the title. Please note which branch it was diffed against (currently all patches should go into the trunk and then to the 1.0 branch, this will change when 1.0 final is out).
> 5. Post followup in the mailing list discussion linking to your Trac ticket.
> 6. (optional) Harass someone to make sure it gets accepted.
>
> (http://docs.turbogears.org/1.0/Contributing)

It might seem strange for the naive spectator that people doing their hobby in their spare time submit themselves to such rules and to such a paradigmatic embodiment of Taylorism like the Trac, where every issue is recorded and tracked and the progress is measured meticulously. But here everyone makes her own free decision whether to submit these rules at all, or otherwise to reinterpret and renegotiate these regulations. Here – at least in an idealistic sense – shared understanding is achieved within democratic spheres of discussion opened up by the structured patterns of use of communication media, so that anyone can – to borrow a concept from Polányi (1962) – align and *coordinate himself to others* if she wants to contribute, or try to persuade others to do so in the case she thinks otherwise. It is generally assumed that everyone would do her best to achieve shared understanding: explicit rules represent the current state of this *self-coordination process*, and they themselves emerge from such processes.

From this perspective, the everyday routine of the FLOSS developer is organized around practices of understanding. She reads the discussions and blogs to keep herself up-to-date in general, and whenever she faces a particular problem, she is expected to search and read herself through the documentation, the discussion archives, the source code comments and the Trac database. If she still cannot interpret the problem within her horizon, she might post a new question on the relevant medium, addressed to the person responsible, framed within a horizon that is already closer to the original

developer than it would be if she had access only the source code. Generally, the answer will not be a fully explicit clarification of the situation: it might refer to previous discussions, internal or external documents and source code locations instead. At the same time, the answer immediately becomes a searchable resource for subsequent discussions, or it can lead to the improvement of the documentation.

4.10. Conclusions of the comparative study

I've chosen my two case studies to represent two opposed ideals of understandability, and two markedly different approaches of achieving shared understanding. These cases also offered a rich source of self-reflection by the actors themselves on the social context of programming language use. The analysis could have been extended to other languages, like, for example, LISP, which embodies other historically situated ideals of understandability, such as mathematical simplicity, economy of syntactical principles, and real-time interactivity (Edwards, 1996: 257), but this would have exceeded the limits of this essay.

In claiming that the technically embodied instrumental environment shapes hermeneutic practices, and the conceived ideals of these practices shape technologies, I'm not arguing for a strictly deterministic connection between cultural context, instruments, and the particular practices of understanding mediated by them. Such a thesis would be easily refuted by pointing out that many open-source projects are written in JAVA, which inherits core features from Ada; the same Cold War cultural context has produced quite different products (like the Internet, which made distributed open-source development possible); and that Python is successfully utilized in hierarchical organizations as well.

What I've tried to show is that *the language framework is the result of the collective interpretation of the general lifeworld situation by programmers, management and political decision-makers. By having been inscribed into the processes of programming language use, this general interpretation permeates the particular practices of understanding that are possible within the language framework.* Hermeneutic practice – whether done by developers or reflected upon by language designers – is thus a central element in software development.

I have also assessed the positions of Heidegger, Habermas and Márkus, according to which our age is characterized by a tendency in which hermeneutic practices are being replaced by technical code. Márkus's explanation seems to be relevant both in the case of Ada and Python, but instead of a given fixed set of shared theoretical assumptions (as he thinks), it is the paradigmatic and specialized nature of *ongoing hermeneutic practice* that grants the shared background, which makes explicit hermeneutics often unnecessary.

Heidegger's and Habermas's position seem to be in accord with the design rationales built into Ada. The problem is that hermeneutic practices in FLOSS projects like Python and Turbogears transcend this horizon. Their visions don't seem to have trans-historical validity: they are only relevant in the case of "Ada-thinking", but "Ada-thinking" has strong alternatives in our contemporary world.

Finally, I conclude: the valuable part of their inquiry is not their particular predictions, but rather *the general framing of their questions*: the emphasis on the praxis-constituting role of language and the central importance of hermeneutic practices in understanding our technological culture.

Function Yes in Ada
(taken from Ada Quality and Style, Ch. 10.)

```
package Terminal_IO is
   […]
   function Yes (Prompt : in     String) return Boolean;
   […]
-------------------------------------------------------------------
with Text_IO;

package body Terminal_IO is
   […]
   -------------------------------------------------------------------
   function Yes (Prompt : in     String) return Boolean is

      Response_String : Response := (others => Blank);
      Response_String_Length : Natural;

   begin  -- Yes
      Get_Response:

      loop

         Put_Prompt(Prompt, Question => True);
         Text_IO.Get_Line(Response_String, Response_String_Length);
         Find_First_Non_Blank_Character:
         for Position in 1 .. Response_String_Length loop
            if Response_String(Position) /= Blank then
               return Response_String(Position) = 'Y' or
               Response_String(Position) = 'y';
            end if;
         end loop Find_First_Non_Blank_Character;
```

Function Yes in Python

```
from sys import stdin

def Yes( prompt ):
    “”” Returns True if user answers ‘y’ or ‘Y’ “””

    print prompt + '?'

    # issue prompt until non-blank responses
    while True:
        response = stdin.readline()
```

5. Software development as social action: distributed cognition or hermeneutic practice?

5.1. Introduction

How shall we render understandable the social-practical processes of software development? Beside the non-systematic, much-debated, yet influential literature of the practitioners' self-reflection (e.g. (Beck, 1999; Hunt, 1999; McBreen, 2002; Boehm, 2004)), there are only few systematic investigations that are based on classical *philosophical* approaches. Given the widespread acknowledgement of the societal relevance of information technologies, it seems strange that philosophical theories don't pay enough attention to the *particular social-practical processes* in which contemporary info-communication technologies *come into being*.

In the following, I'd like to present the theory of distributed cognition (dCog), because, given its privileged position within HCI literature; this is the most likely candidate for a philosophical theory of the social processes of software development. It shares many features of Floridi's PI approach, most probably because of their common intellectual ancestors – most prominently, Simon's and Newell's influential theory of symbol-processing and rational decision-making –, but it also departs from it in significant ways, drawing upon theorists like Brooks, Suchman, Winograd, Dreyfus, Maturana, Lakoff and Norman, who all criticize the rationalistic, disembodied approaches of cognition and computing. (Brooks 1991; Dreyfus 1998; Maturana 1980; Lakoff 1980; Norman 1998; Suchman 1987; Winograd 1987)

I demonstrate with two case studies that the processes, which I call *hermeneutic activities,* lie outside the domain of this theory. These hermeneutic episodes are characterized by the *lack* of a *commonly shared functional description level*, but the existence of such a level is indispensable for the theses of dCog to hold. I claim that the hidden premise that assumes the existence of such a level is not only problematic, but is also inconsistent with the other theoretical roots of dCog. In order to see this, we have to turn our attention toward the practices of *interpretation* that are taking place in situated hermeneutic activities, and we have to handle concepts like "representation" and "information" with suspicion, because they tacitly imply the existence of such a shared functional description level.

In my analysis, I lean on the other branch of dCog's theoretical roots, predominantly on the works of Suchman, Winograd, Dreyfus, and Norman. I'd like to provide support for the theses T1 a.) – e.) outlined in the introduction. Through criticizing dCog, I do not intend to prove its untenability. As we'll see, after a critical reconstruction, dCog is compatible with most of my theses.

5.2. Distributed Cognition

The term Distributed Cognition was coined by Edwin Hutchins (Hutchins 1994). The main tenets of this theory are summarized in (Hollan et al 2000):

> i) A cognitive process is delimited by the functional relationships among the elements that participate in it, rather than by the spatial collocation of the elements.
> ii) Whereas traditional views look for cognitive events in the manipulation of symbols inside individual actors, distributed cognition looks for a broader class of cognitive events and does not expect all such events to be encompassed by the skin or skull of an individual. (Hollan et al 2000)

and these lead to the following theses:

> 1. Cognitive processes may be distributed across the members of a social group.
> 2. Cognitive processes may involve coordination between internal and external (material or environmental) structure.
> 3. Processes may be distributed through time in such a way that the products of earlier events can transform the nature of later events. (Hollan et al 2000)

An interesting application of the theory is the case study of Hutchins & Palen (1997), in which they investigate the interaction between the pilots in an emergency condition during a simulated flight. They engage themselves in a distributed troubleshooting process, systematically searching through the space of possible causes of the instrumental readout. The authors notice, that in conveying meaning, the pilots rely extensively on their shared perceptual access to the instrument panel and on their shared skillful knowledge about possible courses of action. The spatial distribution of the instruments plays a significant role in visualizing inferential steps of the troubleshooting process. Following the gestures and the explanation of the second officer, the pilots shift between different levels of representation, sometimes looking at the instrument as a fuel gauge and then taking it to be a representation of the fuel tank itself. In the first case, they look at the fuel panel, and in the second, they "see through" it. When the secondary officer silently puts his finger onto the fuel test switch, they correctly infer that he means that he has already tested the fuel panel hardware – this shifts the level of representation to the meta-level, involving inferences about each other's knowledge and responsibilities. The authors arrive at the conclusion that articulated speech is just one modality among others: "space, gesture, and speech are all combined in the construction of complex multilayered representations in which no single layer is complete or coherent by itself." (Hollan et al 2000)

5.2.1. The classical cognitivist roots of distributed cognition

The approach of the dCog theorists draws upon the classical cognitivist ideas of Simon and Newell (1976), and they take them further in certain directions. If cognition is *symbol manipulation* and its motivation is *problem solving*, as cognitivists say, why should cognition be narrowed down either to the confines of the individual brain, or to the housing of the "intelligent" computer? Early cognitivism assumed the existence of a single "central processing unit" – but this is only the influence of the technology of their age. Nowadays both computers and people are networked. Symbol-manipulation processes of problem solving are distributed among people and their artifacts. In Hutchins's example, touching the fuel panel test switch externalizes the cognitive process of the second officer: by sharing his belief that the problem does not reside in the fuel panel, he effectively cuts down the dead-ends of the search for solutions in the problem space, thus eliminates redundancies in this parallelized problem solving effort.

The fingerprint of classical cognitivism can also be discerned from other aspects of the example. In Hutchins's description, the features of space, gesture, and speech are already presented as symbolic *representations*, preselected, segmented, and categorized according to their cognitive role in solving the problem. Lines of explanation along other social aspects are not taken into account: knowing that it is a videotaped simulation, it might well be, that the second officer touches the test switch with the intent of *delegating responsibility*, saying, in effect "I've followed the standard textbook procedures and now I'm awaiting further orders." Emotions, social status, power struggles don't play a role in these descriptions. DCog's other examples – ship navigation (Carroll 2003), informal techniques of using the airspeed indicator, the use of the cursor and the "airstrip" in air traffic control (Hutchins 1995; Halverson 1994; Dourish 2001) are all characterized by a narrow cognitivist focus on symbolic or symbolically reconstructable interactions. All their examples consider already well-established, *routine* practices, with a preexisting ontology of action discernible from rulebooks and technical literature. Their description is not a phenomenology of action in a Merleau-Pontian sense, but is rather a functional description of a meta-individual machinery, within which humans play a mechanistic role.

Their approach recently became attacked as one that does not make sufficient distinction between humans and machines. The debate with their answers is summarized in (Susi 2006), but it seems to me, that since dCog theorists share their premises with Simon and Newell, on which the classical argument for the possibility of artificial intelligence is built, they must commit themselves to their conclusions also.

5.2.2. The new waves of cognition: embodiment, situatedness, ethnomethodology

On the other hand, dCog theorists draw on many ideas from Brooks, Clark, Lakoff, Norman, Suchman, Varela and Winograd, who are all very critical toward classical cognitivism. They embrace the idea of *embodiment* by emphasizing the importance of the particular material realization of instruments

and administrative artifacts in our cognitive processes. The airspeed meter with an analog gauge is better than a digital one, because it can be used for perceptual reasoning about relative speeds, and cardinal directions are a more convenient way to communicate speeds than decimal numbers. (Hutchins 1995) The paper file can accommodate such extra information (handwriting, post-it notes, etc.) in an ad-hoc fashion that the badly designed computer database cannot. They also reflect on the *organization of space* as an influental heuristic tool for thinking: e.g., the placement and grouping of computer icons on the virtual space of the screen can encode important meta-information about the files they represent. But their concept of embodiment is still a narrowly cognitivist one: they still take the material realization of tools into account only so far as they exhibit representational capabilities.

They follow Suchman in describing action as *situated* within an unarticulated material background, which serves as a shared resource, upon which all participants of a discourse must draw in order to understand each other. Reasoning, deixis, and disambiguation of words take into account visual, spatial, and cultural metaphors. Their view also endorses Norman's argument, according to which intelligence is not "in the head", but rather in the cultural heritage of the well-designed artifacts, which surround us, and "make us smart", and in the processes in which these artifacts come into being.

The tacit dimension of knowledge or knowledge as skillful action – Winograd's argument against classical representationist cognitivism – is present in the theory, but it does not play a central role.

They also use an *ethnomethodological* approach, which is quite different from anthropological research on indigenous people, since they employ the ontology of their subjects in their descriptions ("actor's categories"), but this "native" ontology is far from being naive: it is thoroughly informed by the legacy of 20th-century administrative science and operation research.

So far there's no disagreement between dCog and my approach. The above aspects are very similar to my theses a.), c.) and d.).

5.2.3. A possible critique against dCog

There's an internal tension between the functionalism dCog theorists are committed to in principle i.), and the emphasis on embodiment and situatedness. If we apply the arguments of Norman, Suchman, Varela, etc. to the concepts of the *analyst*, we find that they also have to be situated and embodied, drawing upon a shared material background, and so on. On the other hand, the universal functionalism of i.) would require an *independent* level of functional description, a level that is applicable equally to all participants of the distributed cognitive process, humans and non-humans alike. Such a functional level presupposes a more or less detailed ontology of action (how the seamless flow of action can be segmented and categorized conceptually into discrete acts) and an ontology of the problem (what are the features that are relevant in finding a solution for the problem, what kind of decision alternatives and computational states can be identified). To build such a functional description, we either have to adopt the concepts of the participants (as Hollan et al. in fact do), or we

have to get involved in the situation. Now, if the descriptions clash with each other, whose ontology shall we prefer?

When dCog theorists talk about multimodally transmitted representations, materially embodied information and so on, these things "in themselves" are nowhere to be found in the flow of action. We can only talk about material processes as instantiating a symbol-transmission when we take the stance of an interpreter, and conceptualize and arrange the action into a functional narrative (a "plan") (Suchman 1987). Talking about "symbols" and "functions" presupposes an act of *interpretation* (similarly to the case of information).

When building a theory of distributed cognition, we have to keep in mind that symbols and functions are not part of a reality that is independent from the participants and the analyst, but rather the participants and the analyst are those, who delineate them and give them meaning. When the horizon of interpretation is standard between participants and analysts – and this is the case in the standard Cog examples, which all focus on well-defined, routine processes – this act of interpretation is generally not reflected upon. But when this horizon is not standard – the use of symbols and their meanings differ among participants –, we cannot assume the existence of a common functional description level. In order to build out this shared level of description, within which they can identify symbols and functions, participants, and analysts have to engage in interpretative activity. *It is exactly this kind of interpretative activity, which falls outside the domain of dCog, because here there are no fixed set of symbols and functions yet*; for solving the problem, the alternatives branches of the search tree, the possible categorizations of actions and computational states are yet to be delineated. Not symbols or representations are transmitted in these interpretational discussions, but the perspective itself, which makes possible the representation of the problem domain and the symbolic definition of possible actions towards its solution.

In the following, I present two case studies, with which I shall show that a great part of problem solving consists exactly of this kind of interpretative activity. Theorists as Norman, Suchman and Winograd are the ones on whom we can build upon, if we want to reach a deeper understanding of this practice.

5.3. Case study: Understanding the Source Code – The Role of Abstraction

5.3.1. Introduction

Software systems and their developers, viewed as a distributed cognitive system, working together to build a solution to a problem domain, seem to fulfill principle i.) of dCog, since there is a given common level of functional description that is shared by developers and computers alike: this is the source code. The programming language has a standard interpretation that maps source code onto the physical operations carried out by the computer unambiguously, and this interpretation is embodied in the compiler/interpreter program. Yet still, the breakdown of understanding between

computer programmers is a major problem among programming language designers (Binzberger 2006). Do programmers really understand source code as a functional description of the problem at hand? Do the symbols of the source code mean the same for all participants? What kind of meta-level understanding do they bring into play when they reflect on and rework the source code?

I show here that the source code comes into being in a process of interpretative activity, and it is subject to further reinterpretation and revision. The current state of the source code, the "development snapshot" is only understandable within a practical interpretative tradition that is constantly rebuilding its own horizon of understanding by improving its conceptual tools and its instrumental environment, and the kind of understanding achieved by them is best characterized in the light of the theses T1. a.) – e.).

There are at least three core features of software development practice, which are central to our investigation.

1. Abstraction is a loose set of practices, through which the particular software solution gets disentangled from the concreteness and contingencies of the use-situation, so that a multiplicity of use-cases gets thinkable and controllable with a finite set of symbolic representations. Abstraction can also be thought of as an "imaginary induction" over the space of possible future use-cases, while looking for lawlike regularities in the projected patterns of use. Modularization, functional decomposition and meta-programming are all examples of abstraction, supported by certain programming languages. The importance of abstraction lies in that it reduces the holistic tangle of relationships of the system and of the problem domain to be thinkable and communicable among developers, sometimes with the explicit aim of "eliminating" their holism. Abstraction makes it easier for the non-initiated developer to situate himself in the background necessary to understand a portion of source code.

2. Embeddedness within a tradition of interpretation refers to the phenomenon that preexisting abstractions of the problem domain – residing in traditional mathematical formulations or natural-language conceptualizations – provide traditional horizons of understanding, which can be relied upon during building a new system. In the case of *metaprogramming*, for example, the goal is to build up a translation between a traditional conceptualization of the problem and the abstractions provided by the system; between the problem-specific language and the system-specific computational tools (Karlsson 2005).

Embeddedness in a tradition does not necessarily mean conservativism. The metaphorical base of a traditional language can undergo great shifts to accommodate new perspectives. In a wonderful article Peter Galison describes how the very first instance of computer use by John von Neumann to simulate the hydrogen bomb explosion introduced such a shift of traditions (Galison 1996). Given the complexity of the simulation, Neumann couldn't have solved his problem with the traditional approach, by the way of symbolically solving differential equations. Having access only to the very limited computational capacities of the ENIAC, Neumann had to find a way to reduce the number of computational steps necessary to calculate the equations analytically. He came up with the idea of

random sampling, borrowed from his earlier explorations into game theory, which later developed into the so-called Monte-Carlo method of simulation. Furthermore, this idea also had the effect of displacing the traditional conceptualization of hydrodynamic problems in terms of symbolically solvable differential equations with a new conceptual tradition, one that laid emphasis on the inherent stochastic properties of atomic processes, and was easier to translate into the language of the available computational tools.

3. Standardization and extending of controlled microworlds. Every abstraction that reduces the complexity of the system achieves this aim by grouping together different use-cases under the same symbolic articulation. This reduces the total diversity of possible situations that might appear in the problem domain by leveling down the differences between them in certain dimensions and accentuating them in others. This "leveling down" can take the form of an adaptation to the structure of the problem domain as it is previously given within a traditional horizon, or it can also play the role of a structuring force that reorganizes the problem domain by introducing similarities and augmenting differences that are traditionally not to be found there. Instead of conforming to the world, this latter form of abstraction makes the world conform to the system, for example, by enforcing standardized ways of interacting with it or by declaring strictly the possible uses of its communication interfaces. To elucidate this point, I borrow Rouse's concept of the "microworld", which is especially relevant here, given its rootedness in AI research (Rouse 1987, 1996: 129, Winograd 1987). Rouse argues that the success of science does not lie in being able to model and predict everything, but rather in being able to *construct and extend microworlds*. Microworlds are reduced models, laboratory settings with finite number of variables that can be controlled and can have their relationships thoroughly explored. In AI research, microworlds are simulated environments in which AI algorithms can be fully tested. The key to success lies in finding a balance between adapting the algorithm or the scientific product to all contingencies of the real world, and between transforming the world itself to bring it into alignment with the conditions presupposed by the algorithm or the product. This is rather well demonstrated by Hounshell's study (1992), who examines two episodes from the history of Du Pont, where this "scaling up" did indeed succeed: the story of the nylon, and the case of the Hanford nuclear plant (the development of the first atomic bomb). In the case of the nylon, the developers faced the task of turning an instable, water-soluble, hard-to-prepare laboratory sample into a stable, water-resistant, mass-producible material, and simultaneously, adapt to existing market demands (women stockings), then leverage this market position to transform and channel various other markets into Du Pont customers. In the case of the Hanford plant, they had to drive up plutonium production from the microgram scale of the laboratory up to the kilograms needed for the bomb. They had to extend the microworld of the laboratory into a plant that at times employed as much as 60'000 people, and simultaneously, they had to stabilize the vast, heterogeneous network of military decision-makers, scientists and technologists by finding ways to align their interests with those of the big project.

To pick out an example of alignment from IT, for a long time it proved much easier to educate people to write standard characters on a touch-screen display, to make them align to the system, instead of devising an algorithm that recognizes or learns various types of handwriting. Adapting the software to the fuzzy variability of the world might address more users, but it also makes it more complex, since it breaks down the commonalities that could be grasped with the same abstraction. On the other hand, standardizing the possible interactions with the system opens up the opportunity of introducing higher abstractions.

From this perspective, following Rouse's argument, abstraction can also bee seen as having an important dimension of *power*. The central question is; how can a handful of developers impose their own abstractions, their own dimensions of similarity and difference on the multiplicity of their users and their use-cases? *How can they extend their microworld into the lifeworld of their users?*

5.3.2. The failure of understanding

The case study is about a market-leader service provider that offers its services worldwide through the internet. My interviewee, George has the task of understanding the source code of the database-management system (DBM) that the senior programmer Andrew has written, in order to be able to improve and modify it. George was appointed to this job because Andrew had a rolling backlog of maintenance tasks and couldn't devote any time to implement new features, and because he started to become a critical risk factor: he alone was able to maintain and develop the DBM. This had an effect also on the firm's market value: a firm, whose critical infrastructure can only be maintained by one particular person, is not worth much in the eyes of potential investors. George's role is thus two-fold: reducing the performance bottleneck and the risk factor Andrew poses.

After a short while, George started to perceive the task as nearly unaccomplishable. The DBM consisted of 4 megabytes of C++ source code, without any documentation and very scarce source code comments. This in itself would not necessarily pose a problem, since there are plenty of such large and badly documented systems (the Linux kernel being one of them), that are intensely developed by a highly decentralized group of developers. But in this case, the source code resembles a cryptogram. Variable- and function names are meaningless abbreviations („a", „p", „prqi", etc.). There are no coding conventions; the source code lines are extremely long. The source code contains lots of repeating patterns. The code is highly redundant: there are at least fifty implementations of the quicksort algorithm, which all differ only in some minor detail. The source code contains many branching points[14] at which the code forks into alternative, unique solutions for each server[15] and service sub-

[14] It is, in fact, a complex, ad-hoc system of embedded code fragments implemented with preprocessor macros (`#include`, `#ifdef`)

[15] The DBM runs on a few different servers.

scriber. All individual settings are hardwired into the source code; there are no configuration files. The abstractions supported by the C++ language are totally absent. There are no objects, templates and namespaces, and there are only a very limited number of function calls. Functions with thousand-line bodies are not rare.

Soon it also turned out, that the situation involves a certain psychological dimension. *Andrew cannot articulate and explain the working of the system.* It is not that Andrew wouldn't know how it works. The system works quite efficiently and reliably, it uses many complex solutions (like an own file system). Andrew can carry out the maintenance and development tasks alone, and he can also explain the working of any particular code snippet step-by-step, but he cannot explain the working of the system in general. He is aware that the system is incomprehensible to anyone except himself, but he argues that the complexity arises from the system being painstakingly optimized and customized to best suit its customers' needs: complexity is a tradeoff for appropriateness and efficiency. But this statement does not fully cohere with reality: the system has reached its total capacity, the firm cannot sign contracts with new subscribers. Andrew is exhausted and no other programmer can help.

It is also interesting that the source code is almost fully independent; it does not rely on any other source code or function library. According to Andrew, "you can only trust what you've coded yourself". Code written by others is incomprehensible, cannot be thoroughly understood, so it cannot be trusted.

According to George's description, the code does not abstract away from the level of the source code: it is actually machine-level (assembly) code written in C syntax. "Andrew doesn't trust code, behind which he can't see the assembly", as George notices. Andrew even checks the code produced by the compiler in the disassembly window. This might improve run-time efficiency, but it has a drastic effect on developer-time efficiency. The individual, idiosyncratic solutions make it impossible to gather similar use situations under a shared abstraction. The improvement on run-time is also questionable. All the fifty quicksort routines are highly customized, but none of them is extremely efficient, because they all implement a naive quicksort algorithm. Instead of fifty quicksort routines, it would be better if there was only a single one that were carefully crafted, well tested, and optimized.

But why is it so impossible to understand this source code? We have an all-inclusive functional description of the system in our hands in the form of the source code, which determines unambiguously the physical operations carried out by the computer. It is also a widespread conception about computer programs that the source code is "nothing more" than the "shorthand writing" of assembly code, and the task of the programmer is nothing else than to reduce the problem to a symbolic functional description (Binzberger 2006). Overall, George's task seems to be nothing else than to do it the other way around: to trace back the variables and the functions to their declarations and initializations.

```
#if defined(MULTI_HASH)
void*HashManager_HM1(void*arg)
{
  if(arg){} // gcc warning
  int32 onwrk;
  Trace(8,"Start HASH comm");
  while(ServerStatus!=1){
    onwrk=1;
    do{
      switch(hprd1->hsts){
        case 2: // Active data, update
          if(hprd1->htbl!=HM_MEMBERID)*hprd1->hptr=hprd1->hval;else{
            if(*hprd1->hptr==0)*hprd1->hptr=hprd1->hval; // Member ID: csak az elsö adatot
kell rögzíteni
          }
          HashReadCnt_HM1++;HashLoad_HM1[hprd1->hidx]++;hprd1->hsts=0;
          if(++HashRead_HM1==HM_PIPESIZ)hprd1=&HashP1[HashRead_HM1=0];else hprd1++;
          RGcoml[R_DM_hPtr1]=HashRead_HM1;
          break;
        case 3: // Deleted data, no update
          HashReadCnt_HM1++;HashLoad_HM1[hprd1->hidx]++;hprd1->hsts=0;
          if(++HashRead_HM1==HM_PIPESIZ)hprd1=&HashP1[HashRead_HM1=0];else hprd1++;
          RGcoml[R_DM_hPtr1]=HashRead_HM1;
          break;
        default:onwrk=0;break;
      }
    }while(onwrk);
    sleepm(2);
  }
  Trace(8,"Stop HASH comm");
  while(hprd1->hsts==2){hprd1->hsts=0;
```

Now let's look at our example excerpt, and see what does the variable `hprd1->hsts` (functioning here as a branching condition) mean? If we run a search against the source code, we find the following initialization:

```
hprd1=&HashP1[HashRead_HM1=RGcoml[R_DM_hPtr1]];
```

This means that the value and the meaning of `hprd1` in the context of the program depend on the variables `RGcoml`, `R_DM_hPtr1` and `HashP1`. If we start to look for their meaning, we find that they are interwoven in a holistic web of relationships: they appear and get modified at many places throughout the source code. Because they are *globally declared* variables, the *temporal order* of these modifications at runtime cannot be reconstructed based on the formal declarations of the functions, within which they get modified. In order to trace back the state-changes of these variables, the temporal order of nearly *all* instances of their modifications have to be reconstructed. Even the simple task of enumerating the values they take up during the course of execution – which is indispensable to find out the states they represent – involves a lot of reconstruction and/or online debugging. This difficulty is multiplied by the various "temporal self-referentialities" that exist in the source: in our exam-

ple, the variable `hprd1` appears in the branching condition `switch(hprd1->hsts)`, which influences its subsequent modification `hprd1=&HashP1[HashRead_HM1=0].`

We can see that the problem of these holistic interrelations is that they result in a situation, where in order to understand one segment of the source code, we have to understand a great deal of it. *Understanding a single symbol presupposes understanding the whole system, but we only have at hand a description of the system that consists of such symbols.*

The interpretation moves in almost Derridean depths: those that are signified by the signs are not present; every symbol is only a deferral, a trace that leads to other symbols. In this infinite drift of semiosis, there is no fixed point to which we could anchor our interpretation. (Derrida 1997). We might even arrive at a "grounding" definition like this one:

```
int32*RGcoml;              // Channel address (int32*)
```

but it doesn't help much, since we don't know what does the word "channel" mean within the holistic context of the system. (It seems to have some very situated and idiosyncratic meaning.) This definition is not grounding, but just another indication, this time pointing forward, toward the web of subsequent modifications and operations carried out on the variable.

Our example boils down to the central question: *what does it mean to understand a functional description?* It turns out that *understanding* a functional description is very different from *possessing* a fully explicit symbolic description. A 4-megabyte long source code that lacks abstraction is almost as meaningless as the original problem was before it was ever programmed. It might be easier to rewrite this code from scratch than to understand it.

At this point, I'd like to recall our thesis a.), according to which understanding unfolds itself in *skillful action*, and that is a certain capability of orienting ourselves in a lifeworld. In our case, George wants to know what happens if he modifies the source code: what other code regions, what functions will be affected? Where should he look for the locus of errors, when errors crop up thereafter? If he is asked to add a new feature, where shall he start to carry it out and what code regions could he build upon?

A good source code is indeed more than a shorthand writing of assembly code, insofar as it aids this kind of orientation, because the symbolic description does not always provide the reader with unambiguous points of orientation. *Moreover, in order to understand a source code line, George does not only have to know its role in the narrow technical context of the system, but he has to know also the role it plays in the holistic context of the system's prospected uses, in the future lifeworld of its users.* This is not a function-attribution carried out at an abstract, conceptual level, but a skill of orientation within the space of the projected use-situations. In order to understand the variable `hprd1->hsts`, we also need to be able to trace it back to users' requirements, and user interface features. This is the orientation skill Andrew can't put into words and can't convey to George. This ineffability is most probably connected to the lack of abstractions, because the linguistic articulation would necessarily involve the grouping together of vari-

ous cases of lifeworld situations. The paradox we face is that *in this case, the symbolic functional description – despite that it is fully explicit – is only an imprint of an undecipherable private language*.

How can anyone orient himself in such a jumble of source code? The lack of abstraction forces Andrew to use special methods. He customizes his development environment with sophisticated coloring schemes, and he uses color pencils on his innumerable notes and sketches. He arranges the repeating patterns of the source code in visually prominent patterns. His horizon of understanding seems to consist of refined skills of orientation within this individualistic similarity space, which George doesn't share, and can't even imagine how it could be approached, because there is no common ground that they could both rely on, and Andrew's private skills resist linguistic articulation. The case is similar to Andrew's natural-language explanations and "specifications": these are just as unintelligible as the source code because they also draw upon a background of skills and knowledge that is not shared by George. The circles of interpretational efforts between them all result in frustrating failures, the convergence toward a shared horizon, appraised by Gadamer as the "miracle of understanding", doesn't start to evolve. This cannot be taken as a fault of the software, but rather as an *acute crisis of understanding* within the socio-technical lifeworld of Andrew and George.

5.3.3. Avoiding the crisis of understanding through abstraction, embedding in a tradition, and the extension of controlled microworlds

The central problem in our example seems to be that Andrew and George approach the source code under different horizons of understanding: in their skillful interaction with the system, they take different marks to be significant for their orientation; in their abstractions, they group together use-situations by different dimensions, and so on. How does this relate to the two other principles mentioned in the introduction: embeddedness in a tradition, and the extension of controlled microworlds?

George's relation to his own horizon is quite different from that of Andrew. George is able to embed and recontextualize his abstractions in the language of traditional approaches. He is able to trace back and reduce his concepts to standard textbook terms, well-known theories, and everyday examples. When he faces lack of comprehension from his peer, because he employs unarticulated skills and tacit knowledge, he can point to books, tutorials, and paradigmatic examples, which can help grasping the necessary orientation. He can also refer back to the standard background of university curricula, so that he can build up a shared orientation with his peers even in special cases when there is no preexisting, shared set of concepts among them. When he labels his variables in his source code, makes his functional divisions, and introduces abstractions in forms of functions and classes, he draws on this public and reconstructable horizon, and he even gives hints for the reconstruction in his comments. To sum up: his functional descriptions and symbolic articulations are carried out within standardized backgrounds; they are embedded into shared traditions of understanding. Whoever shares these traditions can easily understand him. *The good source code surpasses the stage of being a*

"shorthand writing" of assembly code because its creators and interpreters have embedded it in the horizon of various interpretative traditions.

George would expect Andrew to come up with such traditionally embedded explanations, but Andrew was socialized in the pre-internet age, he built up his abstractions in his individual way, and he doesn't have connections to these common traditions. He is suspicious against all abstractions that are not his own. His source code is private and isolated.

George's proposed solution for the DBM would be to use an intermediate formal language, which would make it possible to formulate the problem in a traditional language – for example, by drawing on functional programming idioms or standard DBM abstractions –, but in a way that can be later processed automatically. His preferred solution would be something along the lines of *template metaprogramming* (Karlsson 2005), which harnesses the power of the C++ compiler to define *sublanguages* of C++ that resemble traditional mathematical models or natural-language descriptions; but at the same time, can be compiled and run as any other program.

Andrew's general strategy, "giving unique solutions for unique cases", have a further drastic influence on the possibility of grasping generalities and building abstractions. The extreme size of the source code can be attributed to the fact that all unique solutions involve a duplication of the source code, and then the slight alteration of one copy. In this DBM system, this goes so far that we cannot even talk about "sorting" in general, since there are some fifty slightly different "quicksort" routines, which are all built on different background assumptions and all induce different patterns of use. As a consequence of this, any bug fix or any alteration of the source code draws upon a highly local, system-specific, and situated knowledge; about the particular system on one side, and about the particular use-cases on the other. This situatedness makes the software hard to adapt to a new system architecture or a new user.

If we recall Rouse's model, the key to the success of the natural scientist or the technologists lies in whether he can extend his controlled microworld into the lifeworld of his users. Andrew's strategy tries to achieve this by total adaptation to each user's unique needs. But just what does "controlled" refer to in our case? A software firm does not have a laboratory, in which the experiments can be carried out under a controlled environment, in order to be reproduced! A software firm, on the other side, is a socio-technical system that converts various resources – e.g. human thinking – into software products by producing, using, and transforming source code. In order to be successful on the long run, the firm must stabilize its own code-producing processes. Andrew's idiosyncratic solutions to extend the microworld have a disastrous effect on the stability of this microworld itself: since he is a bottleneck and a critical risk factor, the microworld cannot be extended any further, it cannot be *scaled up* to serve new users and to offer new features. The successes of development are not reproducible, they are bound by the local conditions of the existing users and architectures. And a software within the global market that is bound by such internal constraints of growth is predictable to disappear among its competitors.

Extending the microworld of the software into the lifeworld of its users; but at the same time not breaking the stability of the code-producing processes is a result of a delicate balance. It involves building abstractions, grouping together use-situations and imposing the effects of such development decisions on the users when possible, but it also involves offering unique solutions to unique clients wherever it is profitable and does not risk the long-term internal stability of the firm's processes. Both extremes of the spectrum: extreme rigidity and extreme flexibility in face of possible future use-situations are risky and unstable strategies.

There are other software development methodologies, for example, eXtreme Programming (Beck, 1999), which try to avoid such traps. They emphasize peer review of code ("pair programming"), and a distributed skills and understanding of the source among programmers in the organization ("community ownership of code"). In such a methodology, Andrew would have been forced from the beginning to share his horizon of understanding and his language with his colleagues, and this might have influenced him to use higher abstractions and embed his concepts into the language of some shared tradition.

The relation between the three core concepts is now visible: the value of abstraction and embeddedness in an interpretative tradition are both to be measured in light of the goal of extending the microworld. Abstractions have to be carried out along dimensions that allow for maximizing the covering of the use-situations, while keeping the source code understandable. Traditions have to be followed as far as they do not constrain the perspective of understanding to a conservative tunnel vision. Making good decisions in these matters can only result from a skillful process of interpretation within the context of the lifeworld of the users and the developers.

5.3.4. Conclusion

What's the relation between this case study and the five theses mentioned in the introduction? We've analyzed the kind of knowledge that is necessary to understand the source code, and we've found that (a.) it is better grasped as an implicit skill of orientation, as an ability to find ourselves around in the source code, than an explicit remembrance of the symbols. We've seen that (c.) understanding a symbol is situated as far as it involves envisaging the particular system context and the lifeworld situation in which it gets used. This situatedness poses problems for the stability of the code-producing processes, which can be overcome by employing certain strategies, e.g. by standardizing and reducing the multiplicity of the use-situations through abstraction, or by leaning on and adapting a previously given tradition of understanding (d.). Our case study also shed light on the interpretative processes – characteristic of software development – aimed at understanding the source code and the lifeworld use-situations as well (b.). But so far we haven't talked about the circularity (e.) of interpretation: that is the topic of our next case study.

5.4. Case Study: Engaging the Source Code: Programming as Shared Interaction

5.4.1. Introduction

In this case study, I'd like to focus on the inherent *circularity* (e.) of interpretative activity, and point out the particular relevance of this topic in the case of software development.

In its various formulations, the concept of the *hermeneutic circle* refers to the fact that every act of interpretation is carried out in a previously given, non-articulated horizon of interpretation: understanding a sentence presupposes a holistic system of previous understanding, within which this new element has to be interwoven. On the other hand, understanding a single part might lead us to reassess and restructure the whole of our preexisting fabric of beliefs and intuitive skills, which in turn affects the way we grasp other parts of our understanding. Understanding thus manifests itself in a circular process of permanent revision and reassessment, moving from the individual parts to the holistic whole and then back again.

First applied to the methodology of understanding ancient texts and the Scriptures, then extended into a universal theory about the relation between language, self and the world (Schleiermacher, Dilthey), this idea was firmly established in contemporary continental philosophy through the works of Heidegger and Gadamer (Heidegger 1996; Gadamer 1989) Heidegger's contribution to the problem is the insight that a deeper, ineliminable, ontological circularity exists as an interplay between our understanding of the world and that of ourselves. We understand ourselves through the world – we conceive ourselves in terms of material possessions, naturalistic models, and so on –, but at the same time, we understand the world in the light of our tacit skills, unreflected existential determinations (e.g., the fear from death), and the general background of our socially acquired familiarity of what things mean (Ramberg 2005). Gadamer explores the consequences of Heidegger's idea for the human sciences. He argues that historical texts are not simply value-free objects of investigation, but they are alive, constituting our horizon and forming our world-view. On the other side, we're not passive receivers of this tradition: interpretation is a mutual *interaction* between the text and its reader; the circularity of interpretation is a movement of understanding in which the reader *engages* the text in textual explication and *applies* it within his horizon. Since we cannot reconstruct the original horizon in which it was produced, the meaning of a text is never fixed, but always shifts as it gets interwoven in an ever-richer fabric of interpretations.

Is this theory of understanding confined to the field of classical texts and human sciences? Hermeneutic philosophers of natural science have already shown that this approach opens up interesting philosophical perspectives on scientific texts (Márkus 1987, Heelan 1989) and on the interactions of experimentation (Rouse 1996, Kisiel 2001, Heelan & Schulkin 2003) But so far no one has applied this philosophy to the texts produced within info-communication technologies.

In the following, I show that the interaction between the programmer and the development environment – especially in the case of debugging – could be fruitfully thought of as a hermeneutic circle, i.) in the classical, ii.) in the Heideggerian, and iii.) in the Gadamerian sense.

To show i.), I explain the interaction loops the programmer is engaged in, and the strategies she employs during the constant revision of his partial interpretations about the holistic interrelations of the system and its projected use-situations. To show ii.), I point out that the reconstruction of reasons and causes during debugging is relative to the goals and existential determinations arising from the lifeworld of the programmer and the user. And to show iii.), I emphasize that the programmer's effort is not directed at grasping the system in its technical existence, but rather at bringing closer her horizon of understanding to that of the original programmer, reconstructing her intentions, theoretical presuppositions and conceptual models.

I use Norman's concept of *system image* (Norman 1998: 190) to denote the totality of system interfaces the programmer encounters during his engagement. I use this concept in a wider sense than Norman does: I include the system's various user interfaces, debugging facilities (e.g. log files), the source code, documentation, knowledge bases and further internet-based resources. It is this system image, which is the object of interpretation during software development.

5.4.2. The example situation

Our example starts out as a bug in a small-scale experimental web application that was written in Python, within the TurboGears web development framework (www.turbogears.org).[16] The symptom of the bug is the following: when we use the web-based user interface to modify a person's name that contains accented characters, the modification results in the loss of the characters, which follow the accented one. After the modification, the database contains only a truncated name field. What can be the cause of this bug?

The immediate cause of the bug is obvious. The bug appeared when the application was copied from a Windows XP development machine to a Debian Linux production server. If we reinstalled the application on the Windows XP machine, the problem would be solved. But this analysis is not worth much in our world, given that by the deadline, we have to install the application on the client's Linux web server. Classical theories of causality don't help here much. Then we start to look at the part of the source code, where the accented characters get written into the database. We find the following line there:

```
Name.get(nameid).set(name=name.decode('latin2'))
```

[16] The case study is based on a real, anonymized situation, in which the author took place. The choice of first person plural in the description is motivated by the intention to convey the process in which interpretations unfold.

What does this line mean? Before we could answer this question, let's stop and think for a second, what kind of horizon does it take to understand an answer to this question! My explanation builds on a lay understanding of web forms and databases, a more in-depth understanding of characters, the representation of multilingual character sequences (strings), and the Python language. First, I give a short introduction into these matters. Then I present the case study within this horizon. I will reiterate and organize the philosophical conclusions in the last part, which is readable independently from the case study.

5.4.3. Short introduction

In the following, a multi-layered software system will be explained. The elements of this system are *packages* written in Python, and *function libraries* developed in C. A package contains multiple files, which contain definitions about *class*es. A class can contain *variables* and *functions*. A class can be developed from scratch, or it can be derived (*subclassed*) from a parent class. In this case, the child class *inherits* the variables and functions of its parent. This is very handy for abstraction: common operations don't have to be repeated all over the source code: they can be inherited instead from a common parent class. On the other hand, classes serve only as "templates": in order to use them, they have to be *instantiated*: that is, working copies – *objects* – have to be made about them. Each of these working copies can differ in the content of their variables.

Our problem arises because of a problem in the *coding* of the accented characters. There are many standards for coding character sequences into numeric byte values. latin-1 coding can only be used to represent Western European languages, because it doesn't handle most accented characters. latin-2 is designed for Eastern European languages, whereas Unicode (utf-8) is a world standard that can accommodate all languages of the world. Strings can be converted between different encodings by using the encode(...) / decode(...) functions – at the risk of losing some characters.

5.4.4. Debugging

The above mentioned line is responsible for converting (decode(…))the name parameter that is received from the user interface in latin2 format into utf-8 encoding, and then writing it (set()) into the database. The class Name is subclassed from the SQLObject class that can be found in the SQLObject package. The package implements an object-relational mapping, which means that it hides the complex details of database access behind such simple statements as get(…) and set(…). Deep down in the system, real database access is carried out by the MySqldb package, which is a wrapper around the libmysql.so C function library. The database is stored on a MySQL 4.1 server. For the string to get written into the database, it must pass through all these software layers: first our

application, then the SQLObject and MySQLdb packages, libmysql.so and finally, the MySQL server (residing on the same machine)[17]. The bug occurs somewhere along this path. To us, simple programmers, the details of this process are almost as mysterious as for the uninitiated reader. Writing a program is putting together black boxes: boxes that we're not supposed to open. But in this case, we have access to the source code, so we can.

First, we turn on the error logging function of the TurboGears framework, so that we can see detailed information about the program's inner workings. The logging facility extends the interaction loop of the programmer, so that she can perceive her programmatic environment and follow the results of her actions in much more detail. Without this facility, programming would be a trial-and-error enterprise just as threading a needle blindfolded, without continuous visual feedback. Now, after restarting and reproducing the bug, a relevant error message gets visible between the thousands of irrelevant log entries:

```
mysqlconnection.py:66: Warning: Data truncated for column 'name' at row 1
```

We seem to be lucky: the interpretation of the built-in self-diagnostics is very similar to our formulation of the problem. Let's follow the suggestion and look at line 66 of mysqlconnection.py:

```
    cursor.execute(query)
```

The problem is that we don't know the content of the variable "query" during program execution. We start searching for ways to increase log verbosity: we use Google to find that the SQLObject package has an undocumented, extra error logging feature. If we supplement the database connection parameters with „debug=1", then the content of all database queries will get written into the error log. After restarting, we see the following entries in the log:

```
2/Query   :  UPDATE name SET name = ('AlmĂĄsi') WHERE id = (1)
[...]
mysqlconnection.py:66: Warning: Data truncated for column 'name' at row 1
```

This is the operation, which results in the truncation of „Almási" into "Alm". We can see now that the variable "query" is actually not a query, but can hold any kind of database request – in this case, this is an "UPDATE" request. At this point, the problem seems to be with the encoding of accented characters, but we don't know exactly what is it, since the strange characters („ĂĄ") might have arisen in the logging module. Because both the database handling and the logging module appear to

---

[17] I've omitted the details of the operating system, the network architecture, etc., which also might have had caused the bug.

us as black boxes, over which we don't have the slightest control, it seems rational to turn our attention toward a module, which is the least opaque among them: our own source code. On the other hand, our source code seems to be perfectly okay: it seems to contain a statement, that is coherent with all of our expectations and background knowledge. The problem might be that we've misinterpreted the documentation of encode(). It might also well be, that the framework itself contains a bug. To check either of these cases, we would need to open the black boxes and start to look at the source code of the framework. This is a tedious task. Isn't there a way to avoid it?
What we can do is that we take the whole system itself – database, framework, webserver, etc. – as a black box, which takes our source code as an input, and emits log entries as an output, and we start to perturb the input systematically, to explore empirically the relation between the source code and the log output. In other words, to avoid opening the black boxes, we create a *quasi-experiment*, which determines the input-output mapping of these boxes. In an iterative fashion, we try the following alterations of the source code:

```
Name.get(nameid).set(name=name)
Name.get(nameid).set(name=name.encode('latin2').decode('utf-8'))
Name.get(nameid).set(name=name.decode('latin2').encode('utf-8'))
Name.get(nameid).set(name=decode('utf-8'))
Name.get(nameid).set(name=unicode(name,'utf8'))
[...]
Name.get(nameid).set(name=name.encode('utf-8'))
```

Strangely, none of these versions works. We get thousands of lines of log output, among them various different error messages, but only the last one seems to offer a hint of where to look next:

```
File "/usr/lib/python2.4/site-packages/MySQLdb/cursors.py", line 149, in execute
    query = query.encode(charset)
UnicodeEncodeError: 'latin-1' codec can't encode characters in position 26-27: ordinal not in range(256)
```

Now this is extremely strange. We know that the database expects the query string to be encoded in utf-8. We also know that the name is entered in latin2 on the web user interface. These are relatively hard facts, because there are other applications (written in PHP) running on the same webserver, which work well under these assumptions. But this line mentions the 'latin-1' codec, which we never intended to use at all, since it obviously cannot accommodate accented characters. The problem seems to be that somewhere deep in the MySqldb package, in cursors.py, line 149, an extra character conversion occurs:

```
query = query.encode(charset)
```

We now suspect that this line transforms the query variable – which contains our database update request – into latin1 encoding, thereby truncating our name string. To test this assumption, we make a backup of cursors.py, and replace this line with

```
print(charset)
```

We test the program again, and we find in the log that the value of the charset variable was indeed "latin1", but the program still won't run. Now the name does not get truncated, but instead strange letters („ĂĄ") appear in the database, and on the user interface. Now we start over with perturbing the source code, and we find that the following line

```
Name.get(nameid).set(name=name.encode('latin2').decode('utf-8'))
```

results in perfect operation. We seem to be ready. The cause of the problem was found to be an extra encoding step that took place in cursors.py, and this can be solved by deleting the relevant line. But this analysis is not sufficient. Altering the source code of the framework is very dangerous, first, because we don't know whether our alteration has not broken other parts of the code, and secondly, because the framework will get updated every week or so, and it is not good practice to maintain a version that is different from the official. Later, there might be also other applications running on the same server that are written under the assumption that this extra encoding does take place. And in general, the problem is that our trial-and-error method did result in a highly counter-intuitive solution: we cannot explain, why does the encode()-decode() pair work, whereas the original solution does not?

We have to engage our hermeneutic skills now. The programmer of MySQLdb must have had some rationally defensible intentions in mind when he included that extra conversion step! There must be some reasons behind implementing the most important database access module this way. This extra encoding can be meaningful, for example, under the assumption that it serves the function of some centralized encoding mechanism that is working at odds with our intentions. We turn to the documentation, but it does not mention anything similar.

To disentangle this confusion, we have to employ a different strategy. We have to open the black box (Latour, 1987). Only if partially, but we have to reconstruct to a certain extent the intentions and the tacit assumptions of the original programmer, under which the program was written. In this case, a good starting lead seems to be the "charset" variable itself. Where does it get its faulty value?

Two lines above the problematic line, we find the value assignment of the charset variable:

```
db = self._get_db()
charset = db.character_set_name()
```

This makes some further research necessary. What does the _get_db() function do, and what is the content of the "db2 variable? We use the search function of our file manager only to find the definition of _get_db() in the definition of the BaseCursor class:

```
def _get_db(self):
    [... irrelevant lines ...]
    return self.connection
```

This in turn begs the question: where does the value of the variable "connection" come from? Now our investigation comes to a dead-end. The value of connection comes somewhere from the "outside", during the instantiation (__init__) of the BaseCursor class.

```
def __init__(self, connection):
```

Since Python does not require fully explicit type declarations, there is no easy way to find out the class of the object variable "connection", in order to find the place where it is defined. We have to look for the exact place, where the BaseCursor class or any of its descendants gets instantiated and used. In order to find this, we carry out some other searches on the files of the MySQLdb package.

```
grep –R BaseCursor *
grep –R Cursor *
```

The results all seem to point toward connections.py: this is the only file in the MySQLdb package, where a Cursor class gets instantiated:

```
default_cursor = cursors.Cursor
```

In this case, the instantiation does not invoke the abovementioned __init__ function. We reached a dead-end again, the further searches

```
grep –R default_cursor *
grep –R cursor *
```

do not provide any more hints. Now we might want to continue interpreting connections.py, because it is still highly probable that the variable "connection" is an instance of the class "Connection", which is defined here, but it might prove to be more fruitful to backtrack to one of our earlier questions: what does the following line do?

```
charset = db.character_set_name()
```

No matter what is the class of the variable “db”, it must have a method called character_set_name()! To our great astonishment, the search

```
grep –R „character_set_name” *
```

results only in a list of invocations of the function, without any definitions in it! This function must be defined somewhere outside the MySQLdb package! We repeat the search on the whole Python directory:

```
grep –R "character_set_name" /usr/lib/python2.4
```

While browsing the results, the following line catches our eye:

```
Binary file /usr/lib/python2.4/site-packages/_mysql.so matches
```

After closer examination, it turns out that the function that gives back the erroneous “latin1” character set is defined in _mysql.so, which is a C function library. In tracing back the cause of our problem, we face the task of opening another black box: but given that _mysql.so is a binary library, it would take efforts that are even more intensive. We could get the source code from the internet, but it seems easier to look at the documentation first. Using Google in an iterative fashion, refining our search string step-by step

```
character_set_name()
character_set_name() site:www.mysql.com
character_set_name() site:dev.mysql.com
character_set_name() site:dev.mysql.com 4.1
```

we arrive at the following page:

```
http://dev.mysql.com/doc/refman/4.1/en/mysql-character-set-name.html
```

where we get to know, that the function of character_set_name() is documented as follows:

> **Description**
> Returns the default character set for the current connection.
> **Return Values**
> The default character set

Of course, this doesn't help much. We still don't know where this "default character set" gets erroneously set to "latin1". The interactive comments at the bottom of the page, written by other MySQL users, refer to a bug with version 4.1.8, but it seems so that it was corrected in 4.1.9. Another entry complains about errors in 4.1.10 but there are no answers. Maybe there are still problems with version 4.1.15? We cannot decide.
Let's take a different approach. Let's search through dev.mysql.com for all the possible parameters, which affect the setting of the default character set! We still assume that there is a reason for implementing a function like character_set_name(): there must be an option to explicitly influence its result. Furthermore, this database server is used overall in the world; the developers must have thought of solutions to overcome problems like this.
After a thorough research of the website, we find that the only place to set the default character set is the configuration file my.cnf. If we don't set this parameter, the default value is "latin-1". This is very akin to our problem, but is inconsistent with the fact that on our Windows test server, there wasn't any my.cnf file, and the application worked nevertheless. Since the similarity to our problem is very strong, let's put this inconsistency into brackets, and let's look at my.cnf. In our installation, we find the following relevant lines:

```
[mysqld]

[... irrelevant lines ...]

init-connect='SET NAMES "utf8";'
character_set_server = utf8
collation_server = utf8_general_ci
```

It might be, that these settings are not sufficient to override the default character set. In order to test this assumption, we add the following line:

```
default-character-set = utf8
```

After a test, we conclude with a slight astonishment that the bug still exists. Given the abovementioned inconsistence and this failure, we could go trace back to a previous step, and, for example, download the source code of _mysql.so, but somehow we still feel that the arguments behind my.cnf, based on our general understanding of how people think are still stronger than the manifest inconsistencies we're facing. So we continue in this direction, and start to look for a way to check whether the value of the default character set indeed gets set. After consulting Google and "man mysql", we find two commands that can serve that purpose.

```
mysql -e "show variables;"
mysql -e "show GLOBAL variables;"
```

Yet however, the results show that all character sets are set to "utf8", without exception. There's no trace of the erroneous "latin-1" setting. We still don't know where this setting comes from. We reached another dead-end.

I must admit at this point that my narrative about the debugging process was implicitly structured by norms of rationality. I depicted the programmer as a quasi-rational decision-maker, who carries out what Herbert Simon calls "satisficing behavior". My story could be read as if at each point, there were only a discrete number of alternative paths to follow, and the programmer would weigh arguments for and against each alternative, based on his strictly limited knowledge, in order to choose the path of execution that most probably leads to a solution. Whenever he faces a dead-end, he traces back his steps to the last decision point, where a viable alternative was left unexplored. He searches through the space of possible solutions just as an *A-star* algorithm would do it, using his background knowledge and his human social skills as *heuristics* in the search for the cause of the bug.

However, I've also already hinted at features that are at odds with this interpretation. Foremost, every step involves analyzing and interpreting hundreds of log entries, source code lines and documentation, just to filter out everything irrelevant. This is a highly informed, skillful activity, which contributes dramatically to the reduction of the possible alternatives. In complex environments, it might also involve the use of programmatic filters, helper scripts and log analyzer products. I've also omitted the details of source code perturbation. It takes many experiments, sometimes even writing tiny example programs just to make good, informed guesses about what could possibly work, and if something works, it takes even more experiments to settle upon an interpretation. Each guess results in some log output that has to be combed for relevance, and each change – compile – start – test – analyze – interpret cycle can take quite a few minutes. I've also hidden the meanders we made because we didn't read the documentation thoroughly enough – I've built the "best practitioner" into the narrative. In reality, it often takes hours just to debug and localize an error in our understanding. I've also hidden the details of searching through and "scanning" masses of natural-language communication, looking for traces that could provide us with a new lead. And I didn't explain in detail the numerous other experiments, whose results later turned out to be utterly irrelevant to our goal.

But at this point, we're really stretching the limits of such a rationalistic analysis. Our next rationale step would most probably be to backtrack and download the source code of _mysql.so, since all the evidence suggests that the settings in my.cnf are irrelevant. Alternatively, we might as well carry out the risky change in MySQLdb, create a "patch", and warn the server administrator, that he should apply this patch, whenever he carries out an update. But we still have a strong intuitive feeling that the existence of character_set_name() must have some intelligible reason, and that involves a relatively simple way of altering the default code page. If this is not so, then a bug report should be posted on the dev.mysql.com forum, because it violates our intuitions about how programmers generally think.

At this point, our programmer engages himself in explorative experimentation. We comb through the bug forums of dev.mysql.com, in search of a similar bug report. We pursue the lines of the following logic: if there had indeed existed such a problem, it would have had violated others' intuitions also, and there would be a bug report on it. Since we find none, our intuition about my.cnf gets reinforced. We start to perturb my.cnf, and we try to use other client programs to connect to the database. We make test programs to check the working of MySQLdb, and we alter the source code to produce even more debug output, to make sure that our interpretation is correct. Since we don't have a plausible model, upon which we could act, we can't narrow down the number of possible alternative paths. To save time, we don't explore this enormous space systematically, but by changing many factors in each perturbation at once, we do a random sampling instead. We hope that the effects of our multiple alterations are not too closely interdependent, so our sampling gives a representative picture of the search space, and it could provide us with a new lead faster. We crudely delineate two sets of results: the erroneous ones, in which the application stops with an error, and those, which in fact result in a working application, but the solution – just as in the case of the MySQLdb alteration – somehow does not fulfill our long-term intentions. This exploratory phase lasts so long, as we enter the following command (without any well-defined goal):

```
mysql –help
```

Completely unexpectedly, after the help screens, the program emits the list of default settings. Our attention, attuned to the word "character", focuses on the following line:

```
default-character-set           latin1
```

That means, there is at least one application (the mysql command line client), which uses the problematic latin1 charater set! Where does this program get its default settings? All documentation seem to point toward my.cnf. But we have already added the relevant parameter to my.cnf! We start to read my.cnf very thoroughly again. The following lines spring into our attention:

```
# This will be passed to all mysql clients

[client]
port            = 3306
socket          = /var/run/mysqld/mysqld.sock
```

So far, we have carried out all of our alterations in a different region of the file, under the heading [mysqld]. It might be the case that the client applications, connecting to mysql servers read only this part of the file, and not the one that we have altered! In order to test this hypothesis, we copy the setting

```
default-character-set = utf8
```

just below the heading [client]. The result of `mysql -help` reflects the change, and, after having restored the sources to the state before the perturbations, our application works as originally expected.
We can conclude that the cause of the error was that the default character set, "latin1" was not overridden by a corresponding setting in my.cnf. Thus character_set_name() returned "latin1", which resulted in an erroneous extra encoding step. The reason for this might be that the developer of MySQLdb wanted to be consistent with other mysql client applications, and use my.cnf to provide a centralized mechanism to control the encoding of database requests. But why did our application work under Windows XP, where my.cnf didn't even exist? It seems so, that the Windows version of _mysql.so, mysql.pyd has "utf8" set as default, which didn't need to be overridden.
We still don't really know, what's going on in MySQLdb, but we have a reasonable explanation, a well-defined cause, and a straightforward way to resolve the bug, without altering the source code. We should carry out some further tests now to see that the new configuration setting doesn't break anything else, and that it fully resolves our problem.

5.4.5. Conclusions drawn from the case study

Debugging is the part of software development, which almost never appears in standard textbooks and software development methodologies. Textbooks typically deal with finished, cleared-up source code, and their examples are typically simple and easy to grasp. Nevertheless, in real projects, the programmer faces a different world. Real source codes and systems are complex, badly documented and interdependent in variously unexpected ways. They are generally much larger, than that could be remembered; their function is often unclear; the documentation – that is supposed to reflect the original intentions and presuppositions behind the source code – is often incomplete, ambiguous, incomprehensible or outdated; and the programmer has never enough time to carry out a thoroughgoing interpretation of the source and the documentation.
The symbols of the source code, which was written by someone else, often appears meaningless to the subsequent programmer: she cannot judge, what long-term effects would be induced within her lifeworld by altering them. The source code is not a transparent symbolic representation, but rather a jumble of black boxes, all of which would take a great amount of interpretative effort to peek in. And this black-box metaphor doesn't even grasp the temporal dimension of the problem: in an active project, a great part of the sources undergo constant change; the synchronicity of the documentation and the source code often breaks down; and what now seems to be reliable background knowledge, in a later version might inconspicuously become outdated.

The world that is uncovered in the source code is better conceived as a jungle, with limited line of sight, perpetually metamorphosing figures, hidden dangers lurking behind the foreground of phenomena; where the programmer always faces new situations and new challenges.

How typical is our example of IT in general? One might say that I could have avoided all this debugging if I had precisely known how my.cnf works. I was acting upon a bad conceptual model, in which there were no relation between my.cnf and _mysql.so. But this relation was documented nowhere, and on the Windows-based development machine, my.cnf didn't exist at all, with everything working fine. Beforehand, my conceptual model proved to be correct enough to cope with my tasks.

I argue that our case exemplifies a very common situation in IT. Generally, there are always thousands of parameters, modules and system components, of which we don't even know that they exist, and yet their working is highly relevant for our task. Our black-box approach works under the assumption that we can cope without ever considering the content of these black boxes in all their detail. We only become aware of their existence when we face a problem that proves insoluble in our source code. We're pretty much at the mercy of the original programmer, insofar as she provides reasonable defaults. It indeed takes a great amount of trust to build upon a framework that was developed by someone else.

We can also see that the bug is not primarily a technical phenomenon. It is rather a failure, a mismatch of understanding between two programmers, a difference in the conceptual categorization. To me, it seemed natural that the database access library is independent from the central configuration file: to me, _mysql.so is not a database "client", but for the original programmer, it is. We didn't think of this statement as an explicit proposition, and it wasn't even tacit knowledge: it just simply never occurred to me that there could be a relation between the two. It is only in retrospect, in my rationalistic reconstruction that we can grasp it as a proposition, in order to put it in contrast with the original programmer's reconstructed conception.

I implicitly assumed all the way that I would be able to find not only a *cause*, but also an intelligible *reason* for the bug. I was able to delineate the "cause" of the bug on different levels, in terms of the server machine (the bug was caused by relocating the application to a different server), the character encoding (the bug was caused by an extra encoding step) and the default parameters (the bug was caused by a missing configuration entry), but only the last one could be reconciled with an intelligible explanation. It is not the case that we're talking about the same bug on different description levels, but the bug itself is defined differently in all these cases, and these definitions give rise to different ways of conceptualizing and resolving the problem. Theoretically, we could as well define the cause in terms of assembly code, or electrical signals, but here *the conceptualization of the cause is structured by our possibilities of action and existence* (our lifeworld constraints, our goals and our preexisting instrumental-conceptual horizon of understanding). The reconstruction of causality in these everyday situations is very much relative to our existential situation (ii.).

I also assumed that the variables and statements of the program (the extra character conversion) are not there by accident, but that they are meaningful in the wider use-context of the program module. This assumption governed our decisions; I generated my hypotheses and interpreted the log entries in its light. It never occurred to me, that the names of the variables and the self-diagnosis in the log messages could be misleading. It seemed natural that they are already inscribed with hints about their possible interpretation, and that these hints are intelligible in natural language.

We based these assumptions on our general understanding of how people think and work, the general lifeworld constraints on others and ourselves, and specifically: on our belief that others are likely to introduce meaningful abstractions and general solutions to general problems in quite similar ways as we would. While interpreting the source code, and during browsing the discussion forums, we extensively relied on our "mind-reading" skills: we invoked patterns of reasoning over others' mental states, like "others would have also noticed that …"; we made decisions based on the reconstructed intentions of unknown programmers; we judged tacit measures of similarity and consistency. We acknowledged the problem that others view the same symbols under a different horizon, and we tried to transform our horizon of understanding and bring it closer to the perspective of the original programmer, to understand her concerns and recontextualize them within the relational whole of our lifeworld (iii.).

There is a special, distinct mark of open-source programs that could affect our interpretation. We know that they are developed, read, criticized, and improved by many programmers, all coming from different cultures, with different horizons. Based on this, we can also invoke distinctly social skills and corresponding patterns of argumentation, like this: "a program that survived such public scrutiny is highly unlikely to contain idiosyncratic, meaningless elements". (This is in stark contrast with our previous case study, where we couldn't have made such an assumption.)

The knowledge, what we have arrived at at the end is not a "representation" of the source code. We have rather acquired a skill, a new sense of relevance and coherence in matters of database usage. The kind of knowledge that we employed might seem from the narrative reconstruction as a propositionally encoded, symbolic knowledge, but during the action, we haven't formulated our alternative action plans and our arguments. We acted upon a knowledge that is better described as consisting of various, mostly non-explicit capacities: empathy toward fellow programmers; skills of using software tools; a sense of orientation while getting around in the source code; and most importantly, a practice in generating hypotheses, setting up tests, synthesizing interpretations and tirelessly revising and adjusting them. Only the very tip of this iceberg of knowledge was explicated in our narrative as a chain of explicit arguments. The low detail resolution of our explanation is not a drawback, but an advantage, because its purpose is not to serve as a fully explicit articulation of the error condition, but rather as a means to orient the attention of a fellow programmer – who already shares a great deal of our horizon – in drawing a relative distinction between two cases, that otherwise he would not distinguish.

In our example, the process of understanding was far from being a detached theoretical reflection. Instead of trying to come up with an interpretation and then try to fit it to a static code in an independent step, we *engaged interactively* the objects of the symbolic world of the source code: we experimented with them, transformed them to create situations in which we can test our interpretative hypotheses. Using the logging facility and the source code perturbation, we've set up *interaction loops* to get continuous feedback about the success and failure of our interpretative attempts, to get further hints about where to look next. It was this interactive engagement, which made it possible to detect and pinpoint the errors in our understanding, which enabled us to get closer to the horizon of the original programmer through finding out and articulating the differences among us, step-by-step. From time to time, we encountered setbacks, we had to give up our interpretative attempt as a dead-end, and revise the overall interpretation in light of the insights gained. I also illustrated the interplay between the global understanding, which has driven our hypothesis-generation, and the grasp of the particular details, which refined the global interpretation, or led to its falsification (i.).

During programming, in all our decisions, we face an endless number of possible alternative action plans. The uninitiated user faces the 105 keys that are to be pressed, and is dumbfounded at the combinatorial explosion of possible command combinations. It takes much background knowledge and skills, outside the scope of our present analysis, to reduce the complexity of the situation to a decision between only a few explicit alternatives. The "red line" of decisions and goals, which threads all the way through the narrative is simply *not there* in the instance of action: the decisions, the considerations that lead to up them, and the interactions with the system do not constitute distinct phases, but rather an interwoven flow of interactive action.

The knowledge thus gained is not a replication of the original programmer's mental representations, but a partial and skillful knowledge, that is transformed and embedded into the space of our preexisting conceptual and instrumental horizon. It is distinctly ours, because it is relative to our concerns, situated within our lifeworld situation.

5.5. General conclusions from the two examples

In this chapter, I argued for a revised understanding of software development as social action. I've drawn upon the insights of Distributed Cognition and its theoretical roots, but I've tried to shift the emphasis from the classical cognitivist themes to the topics of situatedness, embodiment, skillful action, interpretation, and tradition, which are characteristic of the post-classical cognitivism of Brooks, Suchman, Winograd, Dreyfus, Maturana, Lakoff and Norman. I articulated my theses in the introduction in five points, and demonstrated them in two case studies. I argued that the case studies grasp and exemplify situations that are typical enough, so that the insights gained can be generalized to the whole field of information technologies.

6. Strategies and tactics of understanding between users and software developers: power dynamics of participation in Open Source

6.1. Introduction

Following the late Foucault's conception of power, Michel de Certeau and Joseph Rouse have developed two theories of how technological and scientific artifacts mediate power relations. De Certeau focuses on the *antagonism* between producers' rationalistic *strategies* of normalizing their products' use-situations, and the multiplicity of creative *tactics,* through which consumers appropriate and reinterpret these products. Rouse argues for a *dynamic model of power*, which conceives it as an interwoven fabric of *micro-power relations* that is undergoing incessant reconfigurations as the established state is contested by emerging centers of power. Both of them attribute central importance to the role of creative reinterpretation in the everyday practice of using technical artifacts.
I demonstrate through a case study that their concepts are very relevant to understand open-source software development dynamics, and particularly user involvement in development (Mozilla Firefox).

6.2. Understanding between users and producers

According to common wisdom, the ideal process of software design would lay emphasis on establishing a shared understanding between customers, users, and developers about their concerns within the problem domain, so that developers can generalize and anticipate future patterns of use. This ideal is best achieved through involving users in the specification, the design, and the testing phase, or even in the whole development process. Various software methodologies advocate different models of user participation, ranging from the exchange of formal specifications to on-site user representatives (e.g. Beck, 1999; Hunt, 1999; McBreen, 2002; Boehm, 2004; Brooks, 1995; etc.), and globally distributed open-source software development projects have driven user participation to a level that wasn't conceivable until recently. This approach would tailor the software to the patterns of *local practices*, forcing the developers to align their understanding of the problem domain to that of the users.
On the other hand, the common reality of proprietary business software development is that it often takes abstract, universal models as its point of departure and forces its users to align their local practices and their understanding – whether willingly or not – to various compelling *global rationalities* embodied in the software artifact. Let's look at briefly some examples just to see what kind of concerns are at play in development decisions.

:: Economic benefits of standardization of the development processes. If the software manages to capture the invariant and universal patterns of user practices, or it can be tailored relatively easily to serve the widest possible range of uses, then it can be sold in more copies while keeping development and maintenance costs down. For the user (and the 3rd-party development community),

knowing that a certain product has a large installed user base, an extensive contributor network, and a profitable company behind it means a certain form of security: it assures that she will get access to the necessary product upgrades and maintenance in the future.

:: Software is often developed intentionally to be a political artifact, aimed at replacing established user practices with new, normalized ones. Since software mediates and influences power relations between various social groups, it can be leveraged to alter these relations according to the interests of the developers or certain subgroups of its users. Apart from situation-specific reasons to do so, there are at least two very general rational arguments legitimizing these efforts:
   1. the economic efficiency ("economics of scale") resulting from standardized practices
   2. the increased predictability arising as a result of normalized processes, because it makes progress measurable, results repeatable, and users replaceable.

:: Users, although being experts in their own problems, are far from being experts at generalizing over the whole range of different users, use-cases and future possibilities of the software, because their horizon is often limited to their particular, idiosyncratic local world. Understanding them has its own costs, risks, and biases, which have to be carefully addressed and minimized, because otherwise the drawbacks might overrun the benefits of participation.

:: Society and technological possibilities regularly undergo unpredictable changes, so tightly adapting to the needs of the current user base is often a sure way to lose the ability to expand into emerging, new markets.

These examples are sufficient to show that software lies in the intersection of the contradicting rationales, arising from the power ambitions of various stakeholders. The actual code is a temporary settlement within the complex networks of developers, retailers, users, customers, between different ranks of managers and subordinates, all trying to bend it in the direction of their perceived interest. Nonetheless, this embeddedness in complex power relations is not unique to proprietary software. Notwithstanding that open source development is often depicted as inherently pluralistic, participatory, anarchistic (Raymond, 1999), or even "poststructuralist" (Truscello, 2003), the sphere of these values is strongly delimited by the above outlined rationales. Even if the free/open source license changes the bias of legal and economic relations between developers and users in favor of the users, the code induces complex dynamics in the socio-economic network into which it is interwoven. Work gets done on and with the code. Development decisions gather momentum and get codified in standardized interfaces, mailing list archives, and policy documents. More or less formalized, hierarchical organizations crop up, "charismatic leaders" emerge, carrying out decisions with severe impact on the future uses of the product.

In the following, I argue that de Certeau and Rouse provide philosophical concepts that can help us understand better this dynamics of power. This inquiry is especially relevant in showing the role of understanding and interpretation in power relations. Then I demonstrate through a detailed analysis

of the Mozilla/Firefox project architecture how these global rationales can be reconciled with a model of intensive user involvement.

6.3. De Certeau: strategies and tactics

Michel de Certeau's famous book, *The Practice of Everyday Life* (de Certeau, 1984) is aimed at dispelling the then-established academic apperception that consumption, from the part of the consumer, means a passive reception of goods created by the modernist production system. In order to counter this view, he calls for a closer analysis of the everyday practices *consumers actually do* while watching television, read novels, cook food, or just walk in the city – practices, which are generally considered too mundane to be reflected upon. This is, in fact, a call to "return to the things themselves" in the way how philosophers of the "phenomenology of everydayness" understood it (Heidegger, 2003; Dreyfus, 1991), and although he never spells this out, his method is indeed a phenomenological one: he gives detailed first-person analyses of the lifeworld as it opens up for the "everyman" in such common situations. It also follows Foucault's methodological program in that it approaches the phenomenon of consumption while "taking the forms of resistance ... as a starting point" (Foucault, 1983: 211) of his analysis.

It is this perspective from which we have to approach the occasions of consumption in order to see just how *active*, *creative*, and *diverse* those phenomena are, which remain hidden in the analyses that focus on the *content* of consumption. To inhabit a living space in the city and understand it as a flow of places (and not as a geometrical space), to customize and tinker with mass-produced technological products [*bricolage*], to divert the working time in the factory to pursue creativity and amusement [*parruque*], or to relive the story of a novel within a private world of fantasies: these are but a few examples of the fantastic diversity of the forms of *active appropriation*, which characterizes the occasions of consumption. These "arts of making" [*arts de faire*] stand in stark contrast with the rational order of modern society and production, and disconfirm the view of the consumer as a passive "receptacle" of manufactured objects (de Certeau, 1984: 167).

De Certeau proposes a view of the relation between users and producers as an ongoing struggle between antagonistic *strategies* and *tactics*. Although he employs a military metaphor, he does not draw his distinction between the participants based on their static economic or social properties, but rather on the much more refined distinction between acting in the possession and defense of an own, proper place [*lieu propre*] (*strategy*), and on the other hand, the necessity to act on the terrain that belongs to the other (*tactic*).

> I call a "strategy" the calculus of force-relationships which becomes possible when a subject of will and power (a proprietor, an enterprise, a city, a scientific institution) can be isolated from an "environment." A strategy assumes a place that can be circumscribed as proper (propre) and thus serve as the

> basis for generating relations with an exterior distinct from it (competitors, adversaries, "clienteles," "targets," or "objects" of research). (de Certeau, 1984: xix)

Whereas

> I call a "tactic", on the other hand, a calculus which cannot count on a "proper" (a spatial or institutional localization), nor thus on a borderline distinguishing the other as a visible totality. (de Certeau, 1984: xix)
>
> The space of a tactic is the space of the other. Thus it must play on and with a terrain imposed on it and organized by the law of a foreign power. It does not have the means to keep to itself, at a distance, in a position of withdrawal, foresight, and self-collection [...]. It does not, therefore, have the options of planning general strategy and viewing the adversary as a whole within a district, visible, and objectifiable space. (de Certeau, 1984: 37)

Strategy is thus the practice of territorialization, of delineating and colonizing a region of the lifeworld of consumers. "It allows one to capitalize acquired advantages, to prepare future expansions, and thus to give oneself a certain independence with respect to the variability of circumstances." (de Certeau, 1984: 37) Tactics, in contrast, is the *mode of operation* of those, who were *deterritorialized,* stripped of their proper place in earlier strategic moves. They use tactics to regain control over certain fragments of their lifeworld.

We can classify the global rationales outlined in the previous section (economics of standardized processes and normalized user practices) as strategic, and the adaptation to local contingencies as tactical. The relevance of drawing this distinction to our inquiry is that these concepts can be employed to analyze the power dynamics of proprietary and open source software development. On the one hand, open source development is commonly viewed as a tactical operation, a "revolt" against the strategic positions of proprietary code, aimed at breaking the dominance of existing systems through voluntary participation of users. On the other hand, we tend to forget that open source developers are also in a strategic position with respect to their users and peers. Although they do not charge the user for the software and they let other developers reuse and modify the code, they do circumscribe and defend a place that is their "proper". The regulation of write access to the code repository, the making of development decisions through personal authority, and the politics of standardization are all very common strategic operations in free/open source.

Furthermore, it is the same modernist rationality, as the one we have mentioned in the introduction, which helps open source developers make their users and peers align to their perspective. For example, arguments against the "replication of effort", engaged to prevent the project from *forking* – that is, preventing it from taking a different course than what the leading developers envisaged – are basi-

cally strategic moves masked as economic rationales, enrolled in the defense of the self-identity of the project.

6.4. Rouse: dynamics of power in the texture of practice

Joseph Rouse uses Foucault's conception of power to set techno-scientific practices into a different light. His interpretation turns on a definition of power given by the late Foucault (Rouse, 1987: 211; 2005):

> [T]he exercise of power [is] a way in which certain actions may structure the field of other possible actions.[18] (Foucault, 1982: 208)

This definition – as simple as it might seem – has many deep theoretical consequences, which are already reflected at by Foucault; but Rouse, in his attempt to carry Foucault's insights over to the domain of techno-scientific research, elaborates these consequences systematically. In the following, I follow Rouse's interpretation in summarizing these consequences.

First of all, this definition distances itself from the commonsensical view of power as a monadic property, possessed by a subject, e.g., a political title or an amount of money that belongs to someone. It is not the scalar amount of money that is important, but rather the field of possibilities of influencing the action of others that are opened up by using it.

Secondly, in many aspects, it is a departure from classical theories of power as well. If we look at some classical definitions, we'll see that they tend to focus on the field of possibilities of the actor who possesses power:

> "Power" [Macht] is the probability that one actor within a social relationship will be in a position to carry out his own will despite resistance, regardless of the basis on which this probability rests (Weber, 1978: 53).
>
> In its most general sense, power is the ability to pursue and attain goals through mastery of one's environment (Mann, 1986: 7).

Even a contemporary theorist like Lessig defines market power in this voluntarist fashion, as an ability to raise prices profitably, without experiencing resistance in the form of a drop in sales (Lessig, 2006: 288). In contrast to these, Foucault's definition brings into focus the phenomenal world of the subjugated, and defines power by its role in structuring such worlds of lived experience.

---

[18] Analogously to the concept of second-order beliefs (beliefs about the beliefs (mental state) of others) – acts of power can be thought of as "second-order actions": actions upon the possibility of future actions.

Other theorists conceive of power at the macro-level of social interchanges, as a kind of substance circulating in society:

> Power is a generalized facility or resource in the society. It has to be divided or allocated, but it also has to be produced and it has collective as well as distributive functions. (Parsons, 1960: 220)

Foucault's definition is much more encompassing in its scope than these. It classifies such a wide range of phenomena as power relations as, for example, educational systems, medical institutions, certain cultural practices, sexual habits, and so on. In fact, Foucault thinks that the *micro-power* relations between parent and child, student and teacher, worker and manager, detainee and prison guard constitute the real texture of society, and these are historically reconstructable from traces of shared practices. The macro-power of institutions, such as the Church and the State is founded upon this already rich texture.

This also means that power is not per se restrictive and limiting: many forms of power are constructive, since they provide ways for people to coordinate their action, to align their behavior to each other. The texture of micro-power relations makes possible the existence of our society in the way we know it.

Since this definition grasps power at the *level of actions*, it also denies that power is a "thing" that has a substantial reality independent from the particular, materially embodied historical processes in which it is manifested. Following Foucault, it is also misleading – although often helpful – to visualize power as a hierarchy, a fixed network of relationships between actors, since these "relations" do not exist apart from the recurring interactions or the structured patterns of practice between them.

Power is dynamic, and *only exists through its exercise*, argues Rouse further (Rouse, 2005: 405). Formal hierarchies are only representations of practice, we might add. Any given hierarchy is only a temporary state, a *dynamic balance of forces* that emerges through the actors' antagonistic actions and practices of resistance. We tend to perceive power in terms of settled hierarchies only because we have grown accustomed to the status quo: but power is always *contested* and always possibly subject to dynamic rearrangement (Rouse, 2005: 406).

Instead of being a relation or a property, power is rather a discernible *pattern of actions* within a network of actors, which is stable enough to be re-identified as the selfsame instance of "power" within the ever-changing historical context. The concept is partly the construction of the analyst: it is his responsibility to delineate groups or individuals and identify instances of power patterns between them in the historical stream of action.

Practicing power is not necessarily conscious. Most of the time micro-power relations are not perceived and reflected upon as such. They are embodied in habitual ways of doing things, in tacit social ideals of normality, in procedures of surveillance and record-keeping, and – most relevantly to our

inquiry – in technological artifacts. Foucault, in his famous Panopticon example (Foucault, 1977: 195-228), lays emphasis on the importance of architecture as a technology for structuring space and visibility in the prison. It is not the individual guard as a person, who possesses power over the prisoner. It is the structure of the Panopticon building itself, with its central inspection tower, which creates a power field insofar as it makes possible the total surveillance of the detainees by a handful of prison guards. Power is there because both the guards' and the prisoners' fields of possible action are preformed by the action of the architect.

Rouse philosophical goal is to show that the modern scientific laboratory is also constituted by, and embedded in a rich, constructive texture of power. On the one hand, the instrumentation of the laboratory is focused on the surveillance, recording, and normalization of phenomena, and most of the experimenter's practices are aimed at bringing partially unknown processes under control. On the other hand, its results – publications, theories, innovations, appliances, substances – thoroughly transform and extend the space of possible actions of all the people in our society (Rouse 1987: 226).

6.5. The common thread of the two approaches and the role of interpretation

We can see that Rouse's and de Certeau's conceptions are reconcilable with each other. They are both rooted in the tradition of the "phenomenology of everydayness", they are both faithful to basic foucauldian notions such as "disciplinary power" and "antagonism", and they also share the methodological orientation, which focuses on the forms of resistance in order to reconstruct the historical discourse (Foucault, 1983: 211). Whereas Rouse theorizes about the practices of the laboratory, de Certeau follows ordinary people in their everyday activities. Doing so, they both face and start to reflect upon the same methodological obstacle that hides the complexity of these practices beneath the surface of theoretical formulations and commercial statistics. Laboratory practices and consumer tactics – in contrast to modernistic procedures or Bordieu's *habitus* (Mander, 1987) – are highly volatile phenomena. Their instances are unique because they are carried out under idiosyncratic conditions, and the significances that are ascribed to them by those who participate are situation-specific. They are defined as tactics exactly by their ephemerality and as soon as they turn into established patterns, they become strategies. Yet, for recapturing the volatile and situated meaning of these practices, we can only use language, which is geared at sharing what is common and universal in our experience. Practices are elusive targets for a theoretical description.

But it's not only the theorist who interprets. "We interpret ourselves and the world in what we do and how we do it; our everyday practices and our bearing as we engage in them make us what we are and repeatedly remake us" (Rouse, 1987: 59) Rouse uses "interpretation" in the heideggerian sense, which is much broader in its scope than the reconstruction of meanings from textual traces. Our embodied existence: our skills, and our habitual ways of doing things already constitute a non-articulated interpretation. Our tools and products are meaningful against the background of this prior understanding, and working out new uses amounts to working out new meanings for an arti-

fact. *User appropriation brings new interpretation to the technical product as it situates the artifact in a form of practice not premeditated by the designer*. The product is inscribed with preconceived normalized use-patterns, but as the user departs from this normality and recontextualizes the product within the horizon of his own world, the essence of the product itself changes with the interpretation.
From this perspective, reinterpretation of texts and objects is a fundamental tactical activity: a resistance against normalized meanings, which are produced in the modernistic production system.

> Reading is only one aspect of consumption, but a fundamental one. In a society that is increasingly written, organized by the power of modifying things and of reforming structures on the basis of scriptural models (whether scientific, economic, or political), transformed little by little into combined "texts" (be they administrative, urban, industrial, etc.), the binominal set production—consumption can often be replaced by its general equivalent and indicator, the binominal set writing—reading. [...]
> The reader takes neither the position of the author nor an author's position. He invents in texts something different from what they "intended". He detaches them from their (lost or accessory) origin. He combines their fragments and creates something unknown in the space organized by their capacity for allowing an indefinite plurality of meanings. Is this "reading" activity reserved for the literary critic (always privileged in studies of reading), that is, once again, for a category of professional intellectuals (*clercs*), or can it be extended to all cultural consumers? (de Certeau, 1986: 168)

> "[L]iteral" meaning is the index and the result of a social power, that of an elite. By its very nature available to a plural reading, the text becomes a cultural weapon, a private hunting reserve, the pretext for a law that legitimizes as "literal" the interpretation given by socially authorized professionals and intellectuals (*clercs*). (de Certeau, 1986: 171)

Thus the dogma of a privileged, "true" meaning – embodied, for example, in scientific textbooks and the disciplinary mechanisms of educational systems – is a strategic power device, engaged in the defense of established intellectual hierarchies. In parallel to that, Rouse is concerned just with such a group of socially authorized professionals. He claims that the scientific publication can itself be conceived as a power device: it exerts power over its reader insofar as it makes her change her preexisting beliefs and the objects that make up her world (Rouse, 1987: 120). A scientific publication is embedded in a power field by being situated in the context of laboratories, funding agencies, investors and potential users. The classical ideal of "disinterested knowledge" seems to be true only because

this context is not directly discernible from the wording and the literary style of the "finished", standardized results of the research process. Many philosophers of science dismiss this context as irrelevant for the actual process of scientific investigation, but many argue that it remains unexplicated only because it is known and taken for granted already too much by those who use these texts (Márkus, 1987). Within this context, the scientific publication mobilizes supporters and proofs, in order to introduce changes to the status quo[19]. Readers change their beliefs in face of plausible scientific arguments not solely because of their interiorized commitment toward truth, but because if they did not, they would fail to hold their position within the power texture of the scientific community.

I do not assess the question whether Rouse's or de Certeau's theory meshes well with Foucault's own historical investigations, nor whether they are a correct rendering of Foucault's conception in all their details. Neither am I able to give such rich phenomenological renderings as de Certeau does. What I can show is that the concepts of these two theories can be adapted for understanding the dynamics of user-developer and developer-developer relations in software development. In software, power is manifested through altering the user's space of possible interactions with the technological medium. Since our modern, technicized lifeworld is largely constituted by what can be experienced and carried out on the screen, this analytical approach seems to be worth investigating.

6.6. A simple example from proprietary software[20]

Accounting software is typically not thought of as a field of creative user appropriation. Just the contrary: accounting has always been used as a disciplinary device by shareholders, management, and governmental tax authorities to practice supervision and control over the economic processes of the firm: to establish *accountability*. Accounting software on the one hand helps to carry out the necessary calculations for that, and on the other hand, it makes its users adhere to the ideal of formalized transactions and processes embodied in its algorithms, so that the control can be maintained. For that, the whole economic life of the firm – the situated human interactions on the shop floor, lively discussions and concentrated intellectual work in its offices, secret negotiations behind cushioned doors – all get reduced to the relatively few data dimensions of economic transactions represented in the software.

Company employees generally don't interact with the accounting program itself[21]. They normally use some domain-specific software to carry out economic transactions, and these systems relay data to accounting[22]. This arrangement has two important consequences from our perspective. First, the

---

19 See also (Latour, 1987).

20 The example is based on discussions with a head accountant of a medium-sized Hungarian automotive service firm.

21 Accounting is nowadays often integrated with domain-specific functions in a single "management information system", but this is not the case here.

22 At least this is the case in the automotive service industry represented in our example.

domain-specific user interface restricts their field of possible actions insofar as it delimits the kinds of the transactions they are authorized to carry out. The more specific the restrictions are, the less likely they will commit accounting mistakes.

Second, their interface is designed in a way that so that they cannot do any transactions "outside" the software: it constitutes an "obligatory passage point" through which they have to pass in their every-day activities (Latour, 1987: 141). In the ideal case, none of their actions can escape the scrutiny of quantified control.

Accounting software and its domain-specific extensions thus constitute the *proper place* of shareholders, tax authorities, and managers. If one steps into this space, he is immediately subjugated to the *strategies* which are embodied in its architecture (Lessig, 2006). Is it meaningful to talk about *tactics* here? Is there at all any place or motivation for resistance?

To answer this question, we have to look at the wider social context in which accounting software operates. The output of the software is a wide variety of data reports: tax statements, benchmarks of economic processes, etc., furthermore, it provides interactive query interfaces for "drilling down" to the specific details of the particular transactions. This output is then reinterpreted by stakeholders and managers within the original context of the firm's life as symptoms of certain processes, and finally, as arguments for or against taking a certain course of action. Since the firm's life is typically embedded in a competitive market that is always in change, thanks to permanent innovation (Christensen, 1997), the symptoms and argument sought for are also shifting. Even tax regulations change. This way the data needed as output changes, and the software must somehow reflect these shifts. This is complicated further by the changes in the work processes, which can turn the domain-specific input interface inadequate as well.

For example, if the software lets the users freely specify the legal status of companies (like "Ltd.") by its abbreviation beside the name of the company, then experience shows that it leads to a combinatory explosion of alternative writing styles ("LTD." "ltd", ...), or even to the omission of this important piece of information. Discipline can be increased by designing the user interface to accept only a fixed set of predefined company forms (e.g. from a drop-down list). On the other hand, (at least in Hungary), this list is not quite stable. Changes in the laws regulating company forms have induced several changes to this list, new company forms have cropped up, and even old ones were have to be distinguished based on finer details of their legal status ("Rt." -> "zRt."). If these options were hardwired in the code, it would need programmer intervention to change it.

The same story could be told on the much more sophisticated level of business processes, document flows, etc. embodied it code. If the software is written in a fashion that it embodies a rigid, fixed strategy, then it is in need of permanent rewrite. Since the economic rationales press against such tight adaptation to the local circumstances, most software leaves ample room for customization, so that it can be used relatively unchanged even after big changes in the firm's life.

How is this permanent adaptation carried out in the case of accounting software? The particular accounting software (FOKA) in our example was written in Clipper in the early nineties (it runs under DOS), but it is still widely used in 2007, after only slight changes in the code. It does not sport a user-extensible database schema, nor a query and reporting interface that could be customized: it is very rigid in the sense that any customization (changing forms, database, reports) need programmer intervention, so it has rarely ever been done. It does have a custom query function, but it can only work with a fixed set of predefined fields. How can it happen that accountants still use it after 15 years?

One obvious answer would point at the highly standardized nature of accounting practices. But in this 15 years, the company underwent several restructurings, and its business model and tax environment has changed extensively. This answer is certainly not applicable in this case.

The answer rather lies in the creative use and appropriation of the software. In our example, the program is heavily limited by the fact that it allows only very reduced methods of record grouping. Invoices, for example, cannot be grouped and summed by their issuing profit center, their product type, etc. On the other hand, for many record types, the software allows the specification of a field called "registry number", which can be filled with an arbitrary sequence of 2 letters and 6 digits. This field can also be used as a query filter.

To overcome the grouping limitation, accountants have devised a coding scheme, in which the potential dimensions of grouping are mapped to registry number digits, thereby achieving grouping by establishing code regions within the code space of registry numbers. The registry numbers of the Suzuki profit center start with S, the Peugeot with P, and so on. There are different letters for new and used cars, service invoices and so on. The conventions of this "tinkering" are formalized in written accounting policies, so where regulation through software architecture breaks down, regulation is shifted into the space of shared social norms (Lessig, 2006). This *tactic* works by *reinterpreting* the data fields of the software, and *then having all other employees align to this new standard interpretation.* This conventionalization goes on in the social field external to the software. This tactic is practiced effectively to defeat the outdated strategy hardwired into the software. The software's function as a disciplinary device breaks down, but this method gives it a surprising flexibility – at least while the eight digits will prove adequate.[23]

---

[23] This example can be likened to DeMarco and Lister's "self-healing property" of human systems, which we have mentioned in our case study of Python (DeMarco and Lister, 1999: 113-114).

6.7. Mozilla Firefox: harnessing contribution

6.7.1. Introduction

Mozilla Firefox is the open-source browser famous for having been able to reach 15% market share by 2007 against industry giant Microsoft's flagship product, Internet Explorer 6.0. It made its first appearance under the name "Phoenix" with version 0.1 at 23/09/2002, and reached maturity with version 1.0 in March 2005. What sets it apart from most other browsers is its famous customizability and "skinnability", which is complemented with cross-platform compatibility second to none, so that extensions and skins are not only relatively easy to produce, but they are also portable between various architectures. As early as version 0.2 (released eight days later than 0.1), an extension management framework was already in place, and people started to develop highly functional supplemental modules right away.

By 2005, extension development became a blossoming micro-ecology of hundreds of extensions and themes[24], driving user-developer forum traffic on sites like www.extensionsmirror.nl up to hundreds of messages each day. Development, deployment, update, and feedback procedures were consolidated subsequently on sites such as www.mozdev.org and addons.mozilla.org, so that even non-professional end-users could safely browse, download, and install extensions to smoothly change their browser's behavior, including filtering out unwanted advertisements, altering the browser's default user interface, or even influencing the rendering of web pages.

Nevertheless, if we trace back the roots of the successful balance between flexibility and stability that now characterize Firefox to the origins of its code base, we will be astonished how extremely difficult it was to arrive at such a balance. The philosophical relevance of making an in-depth analysis of this case is that

:: it exemplifies the important role of active user appropriation in the success of a software product, and demonstrates the power tactics and strategies in play in such an appropriation process (T4)
:: it shows the significance of source code-mediated understanding ("code readability") in facilitating contribution and shifting the balance of power between certain groups of contributors (T1)
:: it shows how diversely technologies can be interpreted by various users and stakeholders, and shows the pitfalls of conflicting interpretations (T2)

6.7.2. History and context

The roots of the Mozilla Firefox story can be traced back to 1998, when Netscapes' biggest competitor, Microsoft, had reached a point of maturity with Internet Explorer 4.0 where it could provide the same functionality as Netscape – except that it was bundled free with all Microsoft operating sys-

[24] Today the number well exceeds 2000

tems[25]. Sensing that it has lost the "browser wars", and two weeks after starting mass layoffs in January, Netscape has decided to go open source with its core browser development, in an attempt to gather outside support and feedback (Zawinski, 1999). Their decision was motivated by Eric Raymond's manifesto for open-source, *The Cathedral and the Bazaar* (Raymond, 1999)[26] and Raymond himself took part in designing the licensing and the release strategy (Raymond, 2000) The overall strategy was that the open-source project would provide a stable, extensible platform, upon which Netscape could build its competitive, customized browser products; and this concept did not change after America Online (AOL) acquired Netscape later in 1998 (Junnarkar and Clark, 24/11/1998).

Nonetheless, the participants of the new project (codenamed "Mozilla" after the movie monster "Godzilla"[27]), faced a very difficult challenge. Some parts of the released code were unfinished, other parts were so badly written that they were almost impenetrable, and some key components were simply missing because they were licensed by Netscape under terms, which did not permit the sharing of the source (Eich, 15/05/1998). All this meant that at the beginning, the published Mozilla code did not compile, and even when it started to work, it was very hard to recruit outside contributors to the project. The majority of contributors still came from the ranks of Netscape, and they were carried by the same inertia of short-term interests that led to the demise of the Netscape browser (Netscape == Mozilla, 24/08/1999; The Navigator Window UI, 16/12/1999).

A strange kind of struggle has started to take place between Netscape and the new Mozilla.org contributors for the ownership and control of the code. The "new" developers (many of them on the payroll of Netscape) argued for a complete rewrite of the web browser. This step was deemed necessary because the old Netscape source code was extraordinarily buggy, messy, and unstructured. It was such a monolithic tangle of code, it seemed impossible to go on improving it anymore (Spolsky, 06/04/2000; Zawinski, 1999), and furthermore, because user interface data and formatting code was intermingled, it was very hard to localize it as well (Elliott, 21/05/1998). The pro-rewrite arguments gained momentum by the middle of 1998, when Netscape has released a new browser engine, codenamed Gecko (Raptor) (Hickman, 15/04/1998). It was the result of a major redesign: it was intended to remain competitive and extensible for a long time. It consists of a rendering engine (NGLayout) responsible for displaying web pages and user interface elements, and a toolkit for cross-platform extensibility (XPToolkit). This latter comprises the core technologies (XUL, XPInstall) that are now responsible for the extensibility of Firefox (Trudelle, 1999).

The replacement of the rendering engine induced an almost complete rewrite of the user interface and all other accompanying code. Ever since then, many have decried the foolishness of this decision

---

25 I'm not going into the details of the widely publicized Microsoft-Netscape conflict and the ensuing anti-trust case.

26 The book was available online before the printed edition.

27 This was the original internal name of the Netscape Navigator line of products.

to rewrite the software from scratch (Spolsky, 06/04/2000; counterarguments in: More Mozilla RC1 Reviews, 25/04/2002). The rewriting and the subsequent period of the "struggle for independence" of Mozilla.org were characterized by uncertain schedules, confused priorities, and no software shipped to end users (Eich, 26/10/1998; Goodger, 06/02/2006; Raymond, 2000; Zawinski, 1999). Netscape 5.0 was never released. The rewriting introduced two years of delay, during which Netscape has almost totally disappeared from the browser market. Finally, when Netscape 6.0 shipped in 2000, it did not provide groundbreaking new functionality, but was rather loaded with bugs characteristic of immature software (many of which pertained to the usability of its "skinned" interface), and lot of unwanted advertisements of AOL's services (Spolsky, 20/11/2000). User forums, reviewers, and usability professionals condemned it as a total disaster (Matthew, 26/04/2000; MacInTouch Reader Reports: Netscape 6, 14/11/2000; Modern Skin Development Update, 22/08/2000; Major site may no longer support NS, 12/04/2000; Toolbar buttons have a horrible grey bit at the bottom, 27/03/2000).

This presentation suggests that the open-sourcing and the subsequent rewrite decision were among the most important causes of Netscape's failure. Two years of development have lead to a seemingly inferior product. But on the other hand, Firefox developers have started out from the rewritten rendering engine in 2002, and produced a fully functional cross-platform browser with many groundbreaking new features in less than 3 years, with much less resources than Netscape did (Goodger, 10/7/2004). What did they do differently?

The reasons for Netscape's failure might lie elsewhere. An early member of the Mozilla team, Ben Goodger points at the conflict of interests between Netscape and Mozilla, the scarce flow of information between various company departments and Mozilla.org, the "stranglehold" of internal Netscape power hierarchies over Mozilla operations in contrast to the excess autonomy of sometimes unprofessional Netscape engineers, the general lack of professional design focused on user experience, and finally, the immediate market pressure imposed by Microsoft, Yahoo and other companies, under which they had to act (Goodger, 06/02/2006; see also: Netscape == Mozilla, 24/08/1999). At some cases, it was a project management nightmare, with unclear responsibilities, goals, and many conflicting constraints and schedules set by Netscape. Instead of clearly defining priorities and addressing limited targets, development continued on all fronts, resulting in a feature-bloated browser/mailer/news reader/webpage editor suite, which was almost as hard to maintain as the old version before the rewrite. Development efforts dissipated and produced a mediocre result that gathered much criticism and failed to please its intended audience.

A bit paradoxically, in July 2003, when AOL (then the owner of Netscape) terminated the remaining Mozilla/Netscape development team after signing a settlement with Microsoft (effectively pushing AOL users to use Internet Explorer), many have *appraised* the decision as the necessary final step toward Mozilla's autonomy (AOL Cuts Remaining Mozilla Hackers, 15/07/2003). After the Netscape 6.0 fiasco, many thought that the contribution of Netscape in the development in fact hindered the

open-source Mozilla project, as it instilled alien concerns and user interface concepts on it. Mozilla development – continuing under the banner of the non-profit Mozilla Foundation – finally acquired the freedom from such short-term commercial constraints. (We will revisit this issue later.)

By that time, the first blocks of a new open-source browser had already been laid. In 2002, after a few abortive attempts, David Hyatt initiated the Phoenix (now Firefox) project. Apart from being a proof-of-concept technological prototype aimed to leverage the power of NGLayout and XPToolkit, it was also an experiment with a different style of management, and a forceful attempt to break free of Netscape's "deadly embrace".

6.7.3. Tactics and strategies

At this point, the pattern we can discern is that the line dividing the *proper place* of Netscape developers and Mozilla.org staffers gradually shifted and the source code slowly slipped out of the control of Netscape's management hierarchies. Netscape sometimes managed to put pressure on Mozilla module owners to bend the code to its own priorities, but with time, this has met increasingly stiff resistance. (Goodger, 06/02/2006; The Navigator Window UI, 16/12/1999; Netscape == Mozilla, 24/08/1999). Relatively independently from Netscape's commercial failure, the open source community managed to inscribe the code with their own values, and just around the time financial support from Netscape (AOL) ceased, the architecture was ready for contribution-oriented development.

One of the key moments in this process is the Gecko (NGLayout/XPInstall) rewriting decision, because it changed the balance of power in several aspects. First, by having made the source code *comprehensible*, it lowered the entry barrier to the project, and thus encouraged the recruitment of outside contributors. Especially important here is the decision to use very common languages (XML (XUL), CSS, and JavaScript) for programming the user interface of the browser, thereby opening up the possibility of contribution for a much wider community than specialized programming languages would have permitted it. Second, it made the browser strictly standards-compliant, and this empowered web developers to use it as a platform for their web services and applications. Third, it introduced a flexible modular organization – user interface skinnability and extension modules –, which helped to separate the Mozilla browser core from Netscape's layer of commercial customization, and later it enabled Firefox extension writers to integrate their components smoothly with various browser versions and hundreds of other components. (Trudelle, 2/9/99(1); 2/9/99(2); Brendan, 15/05/1998; 26/10/1998)

Before going into the details of each of these rationales (which is the topic of the next section), the interesting point to note here is that most of them don't correspond to the immediate needs of Netscape, but rather to those of contribution-driven open-source development. They were legitimized toward Netscape management as if they would blend seamlessly into corporate strategy, but retrospectively, these can be seen as tactics, initially aimed at gaining ground for open source, and they've

gradually turned into elements of the self-sustaining strategy of Mozilla.org, employed to delimit and maintain its own "proper" space. This interpretation can be argued by pointing out that in 1999, it seemed to defy economic rationality that under the heavy competition imposed by Internet Explorer, developers should focus on abstract goals, like these:

> 1. Make cross-platform user interfaces as easy to build as web pages.
> 2. Support the common UI needs of applications like Navigator, Ender and Messenger.
> 3. Minimize the amount of platform-specific work required to build user interfaces.
> 4. Provide support for customizing user interfaces, e.g., via 'downloadable chrome'.
> 5. Engage the net to contribute via mozilla.org, bringing massively parallel development to XP UI.
> 6. Allow for shipping new products this summer.
> (Trudelle, 2/9/99(2))

In the light of our a posteriori knowledge, we can state that goal 6. was overly optimistic: the first actual shipment took place two years later, with the ill-fated Netscape 6.0. The long term benefits of goals 1.-5. were to be reaped only after 2002, in the Phoenix/Firefox project, well after the termination of the remaining Netscape staff, under a very different open-source development regime.
Even more important is the fact that ever since the beginning, the Gecko framework was perceived by its developers to be a vehicle to promote the practices, values, and ideology of open source. Calls for involvement with inciting lines like these were not uncommon:

> "Adam doesn't have a netscape.com email address, but he's an important part of the NGLayout team. You can be one too by contributing some code, testing efforts, or even just ideas about where we should be going with the project."
> "By the time NGLayout hits the wire for real, it should be the greatest HTML layout engine on the planet. With your help, version 2.0 can change the world." (Gessner and Davis, 25/09/98)

The extent of the contradiction between the open-source practices inscribed into the architecture of Gecko, and Netscape's corporate efforts of "monetizing" the browser can be exemplified by a closer analysis of the debates surrounding the ill-fated Netscape 6.0 user interface (UI).
At the end of 1999, before the release of 6.0, the AOL/Netscape development team started to design a new user interface to leverage the long-term customizability concerns that went into the design

of the Gecko rendering engine. The new "Modern" skin featured a characteristic blue color scheme, round buttons, and rounded corners, all designed for a perfect fit with the new AOL/Netscape portal. This portal was set as the default starting page, and the user interface was also loaded with links and references to AOL content, in the hope that the browser would generate revenues for Netscape this way. An option was offered to switch back to the "classic" skin, but it was relatively hidden away for the unsophisticated user.

The marketing rationales of this user interface redesign seemed obvious at that time. Whereas most of the improvements of the rewritten browser (e.g. better standards compliance, modularity, etc.) were invisible for the average user, the astonishingly new "user experience" (German Bauer) was intended to set the browser visibly apart from the previous Netscape versions and the competitors as well. However, the Mozilla community was deeply stirred by this development.

First, the new skin introduced changes at deeper levels than the Netscape UI customization layer: in order to display AOL content on the browser interface, it required changes to the commonly shared original Mozilla code. Even if the links to AOL content (the so-called "whore bars") were removed from the open-source Mozilla builds, it still had to share the empty toolbar spaces and the general color scheme, thereby making the Mozilla interface dependent from AOL's marketing goals. This arrangement also froze certain changes to the Mozilla browser interface. This conflict of power can be illustrated by the debate in which Dave Hyatt's proposed improvements to the Mozilla/Netscape UI were rejected by German Bauer, the Netscape engineer responsible for user interface issues, working in the Mozilla project (The Navigator Window UI, 16/12/1999). Even after all the core members of the Mozilla community expressed their support for Hyatt's minor, yet well-argued improvements, Bauer did not revise his decision. However justified Bauer's decision was in the hurry before the 6.0 release, non-Netscape-sponsored open-source contributors looked upon such conflicts as Netscape's intrusion into the autonomy of the open-source community. They tried to react by designing a new skin called "Aphrodite", but given the lack of resources compared to those of Netscape, they did not proceed fast (Should Mozilla Have it's Own Skin?, 21/06/2000).

Second, most of the participants in various Mozilla-related developer and user forums found the new skin appalling and hard to use (Thomas, 26/04/2000; Toolbar buttons have a horrible grey bit at the bottom, 27/03/2000; New Skins coming to Mozilla, 22/10/1999; The Navigator Window UI, 16/12/1999; Modern Skin Development Update, 22/08/2000; Salon on Mozilla, 12/03/2002). Some ventured even so far as to file a bug report against Mozilla to use the "Classic" skin as the default (Bug 48205 – Use Classic Skin as default, 09/08/2000). Their recurring point of criticism was that the "Modern" theme abandoned the familiar UI concepts of each target operating system, so users would have trouble finding their customary ways of doing things. For example, the "Modern" skin did not use the conventional 3D beveling around buttons to offer "pushable" affordance to the user, and it used completely new keyboard shortcuts instead of taking over those of the industry-standard Internet Explorer (Spolsky, 20/11/2000). Bauer dismissed these concerns by downplaying

the importance of existing UI conventions, arguing that the cross-platform consistency of the browser is more important than the consistency with the operating system[28] (NUI, 11/12/1999). He did not answer arguments, which pointed out that cross-platform consistency matters only to a tiny fragment of users, and inconsistency with the operating system means a steeper learning curve for most other people.

Totally misapprehending the motivation of open-source developers, he framed his decisions superficially in the corporate marketing lingo of "permanent innovation" and "user-centeredness"[29], instead of giving rationally argued answers for the substantiated complaints. Instead of responding to criticism, he positioned himself and his team as "revolutionaries", in contrast to the more conservative Mozilla.org developers. Since Bauer presented his decisions as the "official Netscape position", his lack of substantial answers to the questioning of his decisions seems to exemplify Goodger's claim about the power conflicts between Netscape employees and Mozilla.org participants (Goodger, 06/02/2006).

Finally, the complaints of Mozilla hackers were countered by the authority of Netscape's "Usability Lab", led by usability specialist Lakespur Roca. Suggestions of veteran contributors of the netscape.public.mozilla.ui newsgroups (like Matthew Thomas, Dave Hyatt and Ben Bucksch) were dismissed on the basis that being professional developers, they only represented the concerns of a tiny minority of users, whereas the data from a usability test with 5-7 people is much more unbiased (Usability Studies at Netscape, 05/08/2000). Let me point out that this argument stands in genuine contradiction to Bauer's intentions, since "cross-platform consistency" is a prime example of an issue, which concerns only a tiny minority of users.

Since the results of these usability tests were not made public for the Mozilla community, the decisions based on them – like those of German Bauer – were not open to rational criticism (see Matthew Thomas's post in Usability Studies at Netscape, 05/08/2000). This introduced a novel asymmetry between the developers, and another contradiction with the mentality of open-source projects.

So far, we have seen how a subset of Netscape employees, by using their professional authority and their privileged position in the Mozilla community, tried to bend the open source code toward their perceived interests: they managed to extend the boundaries of the proper space they kept under control. *Customizability, which was originally intended to empower users in the tactics of user interface appropriation, was enrolled as a component of the corporate "monetization" strategy, aimed to take over the users' online experience with an "integrated" AOL-branded marketing world.*

---

28 "I claim the operating system specifics will matter less and less over time." (NUI, 11/12/1999). This was a very bland underestimation of Microsoft's market position. Most users used and still use Windows as their single platform.

29 "[W]e are going the right path and being commited to innovate to create delightful, exciting and easy to understand interface." (NUI, 11/12/1999)

Their triumph did not last long. The strategy to colonize the users' online world, while ignoring the feedback from the professional community backfired severely. Roca's argument against taking into account the opinion of the minority of professional users might have been valid in a world where the voice of the professionals would count as equal to others, but in IT, exactly the opposite is the case. The product's usability failed to please one of its most important audiences: professional review writers. The bad reviews brought up the very same usability issues veteran developers were complaining about, and the AOL marketing elements of the interface were also met with scorn (Thomas, 26/04/2000 summarizes many of them; Spolsky, 20/11/2000). Usability was just only one of the product's deficiencies, beside its bugs, freezes, and missing features, but these might have been interpreted with more tolerance by professional opinion-leaders, if their interface concerns were better taken into account.

The influential article by (Knauss, 10/04/2000) went a step further, and criticized the central goal itself of the Netscape rewriting effort: customizability. He used Mozilla's deficient usability as an example to illustrate the general failure of skinnable/themeable interfaces. He argued that instead of a customizable interface, what most users need is a *well-designed* one, which conforms to common user interface conventions, retracts itself into the background, and helps them to get their work done. He puts forward an elite, professional designer "dictatorship" as the model to be followed. In the final analysis, it turns out that he wasn't right (the customizability of Winamp and Firefox turned out to be very successful, and nowadays all major operating systems have a themeable interface), but it will take further analysis to show just under what conditions can be customizability successful.

Why did open source volunteers let this happen? Why didn't they put up more solid resistance? First of all, Mozilla.org members did not have much influence over Netscape's use of the code. Most contributors were on the payroll of Netscape: direct conflict with Netscape management would have cost them their jobs, and Netscape would have hired other people. Since the code was open-source, they could have forked[30] the project, but then they would have lost all the benefits of the joint effort, and it would have been very hard to gather the critical mass of volunteer contribution to continue without the support of Netscape. On the other hand, to a certain extent, they did try to resist. Module owners[31] attempted to refuse patches, which were deemed being of substandard quality (Goodger, 06/02/2006), and at last, the Mozilla interface module was forked. Nevertheless, resistance turned successful only after the forking of the Phoenix project.

---

30 "Forking" means here the case when the open-source community "splits up" and the project continues to follow two different development trajectories.

31 Designated person responsible for a code module. The code cannot be changed without his approval.

6.7.4. Rationales for Gecko revisited: architecturing involvement

We have already mentioned that code comprehensibility, flexible modularization and standards compliance were crucial arguments in the NGLayout (Gecko) rewriting. Why are these rationales exactly so important in an open-source project? In this section, I want to show that these are the key *strategies* engaged to reconcile user involvement and contribution with the global rationales weighing against them, which were mentioned in the introduction. These strategies are employed in the Mozilla project in conjunction with a strict command-and-control hierachy in order to normalize development processes and use-situations, while opening up a wide field of creative user appropriation at the same time. They also help avoid the risks arising from tightly optimizing the browser to the local practices of particular user communities, while still offering them opportunities to customize the browser to their liking.

Comprehensibility as an objective appeared first in the Mozilla project in 1998, with the decision to rewrite the browser on the basis of the NGLayout (Gecko) rendering engine. The first goal of the rewriting was to convert the cryptic, monolithic C code of version 4.x into cleaner and more modular C++ code. For a private company, comprehensible ("readable") source code translates into easier maintenance, lower personnel costs, and lower risks, as I've argued in another case study. In an open source project however, comprehensibility means a lower entry barrier and a less steeper learning curve for the new contributors. It helps attracting them also by opening up to them valuable opportunities to learn interesting things, and by arousing a certain fascination with the technical beauty of the source.

The rewrite decision introduced a completely new technology as well. In contrast to conventional applications, where the user interface code is built in the same language as the core application logic, the rewritten Mozilla browser uses a separate set of languages to do that. The rendering engine is responsible for the rendering of both HTML content and the browser's visible user interface elements (the "chrome"). The description of these interface elements takes place in standard CSS, Javascript, and the XML-based XUL markup language. This division separates presentation (XUL, CSS) and interface logic (Javascript) from operating system-specific rendering algorithms (C++). This modularization has at least two important consequences. First, the browser becomes more portable, since only the rendering engine has to be customized for each platform, and secondly, the browser becomes highly extensible and user-customizable, since XUL/CSS/JS editing does not require the complex programming skills and development tools that are necessary for building C++ applications. In fact, customizing the browser is not much more difficult than classical web development in HTML/CSS/JS: one can simply look at and alter the browser's CSS and JS files, and the results can be seen almost immediately. The working of the browser is transparent and comprehensible to a wide range of users.

Retrospectively, it seems a bit curious why it took more than four years to open up this enourmous field of possibilities to end users. The prerequisites of extension development were ready by the end of 1999, with the implementation of the XPInstall ("cross-platform install") toolkit, but back then, it was used exclusively to facilitate the downloading and installing of the Mozilla/Netscape suite. The extension installation framework, that enabled users to share their extension packages, appeared only at the end of 2002, with Phoenix 0.2.

We can compare this model with the approach of industry leader Microsoft. Starting from version 4.0, Internet Explorer has also been very extensible through its Explorer Bar API[32]. This enabled 3rd-party developers to extend the browser's functionality with access to services like web search, online dictionaries and so on. The API documentation was disseminated through Microsoft Developer Network, and many resources helped extension developers to start their projects. At first glance, this looks similar to the case of Firefox, but there are very relevant differences.

The first big difference from Firefox is that to build an extension, one cannot start by just tinkering with the interface: she needs solid C/C++ programming skills and commercial software development tools. These requirements significantly increase the entrance costs of any extension development: only professionals and firms could enter the customization arena. It also differs from Firefox in that the default browser user interface and behavior could not be overridden: only certain menu extensions and interface elements (toolbars) can be implemented.

IE extensions were native Windows executables, which meant that their working was not transparent or comprehensible to users at all. More often than not, they were also closed-source, and came with a license that forbade users to observe what they do (for example, by disassembling them). In fact, after installation, they could execute arbitrary code on the user's machine. In contrast, XPInstall packages normally contain only plaintext XUL templates, Javascript, and tiny binary fragments, which can be reviewed for blatant security or privacy breaches by any moderately skilled user. On the other hand, this openness makes the ideas implemented in them easy to reuse in other projects (which does not align well with the copyright interests of many 3rd-party toolbar developers).

Another important difference is that IE extension developers and users do not form a community comparable to www.mozdev.org or www.extensionsmirror.nl. In contrast to these, for IE, until the start of Windows Marketplace in 2007, there was no centralized forum to coordinate and review the extension-development activity. Even if a user discovered that an IE extension contains malicious software or unsolicited advertising, she couldn't post a message on a central forum, or give a bad rating to warn off other users. There was also no centralized update mechanism, which implied that there was no easy way for extension developers to push security updates to users.

---

[32] Advanced Programming Interface

To sum up: *extension development for IE is not the involvement of a user community* in the sense Firefox extension development is. It is rather the professional engagement of a privileged few. *Only this privileged few has the opportunity to reinterpret the browser* – as a search tool, a news reader, or a linguistic workbench in normal cases; or as an unsolicited advertisement engine or a private data harvester in worse scenarios. The Explorer Bar API constitutes a power field insofar as it enables these developers to structure the future actions of their users, and as far as it delimits the users' opportunities to resist against the unwanted side effects of this influence.

This comparison shows that the comprehensibility of the browser interface code is crucial for the possibilities of reinterpretation. Instead of restricting its users, XUL technology *empowers* them by opening up new ways of experimenting with innovative interfaces, functions, and new possibilities of interaction; and it gives them back the possibility to decide what appears on their screen. *Comprehensibility is an ambivalent form of strategy, because it empowers antagonistic tactics in order to divert and recruit them in the defense of the proper place.*

Apart from increasing comprehensibility, modularity can also help to balance local and global rationales. As I argued in the previous chapters, modularization is not only a cognitive strategy, aimed at hiding the complexity of irrelevant details from programmers, but it is also an organizational strategy to mirror command hierarchies and delineate regions of responsibility in the code. In open source, where there are often big variances in the reliability of the code from various contributors, modular architecture acquires a different function. It helps to isolate and resolve quality problems by making it possible to turn off or replace individual modules. In this sense, modular architecture in open source has a role similar to that in classical software development: it reduces the risk each module poses for the project:

> [The] extension system [allows] for research into new areas without affecting the core and to allow for techies, early adopters, web developers and other specific communities to customize their browsers to suit their specific needs without affecting usability or download size for the mass market. (Mozilla Firefox Development Charter)

In addition to that, replaceability also creates competition between module developers. Forking a module and duplicating some development effort is much more likely to prove beneficial than forking a big, monolithic project, just as the perviously mentioned Mozilla/Netscape interface fork illustrates. In this model, users cannot be forced into disadvantageous power arrangements. Because of the competition, an extension with awkward or "Trojan" features cannot survive for long. The simplest strategy to circumvent the limitations of an extension is to replicate its features and merge them into a more coherent, more usable module. This is very straightforward, given that extensions are licensed in a GNU-style open source scheme, so reusing code snippets is not forbidden.

At addons.mozilla.org, the number of installations is published for each extension module, so the actual state of the competition can be reliably followed. Users "vote" through downloading and installing them, and it is also possible to rate them and give feedback for the developers and other users. Sometimes modules with similar functionality survive for a long time, and sometimes the balance tips toward one of the modules, and as it starts to get the greater share of attention, the development on the other one is discontinued.

What is the relevance of standards compliance for an open-source web browser? How did web standards – the embodiments of normalized practices – got reinterpreted as symbols of "freedom" and resistance during the browser wars? While end users often don't care about web standards when choosing browsers, web site developers – a different group of browser users – depend on them in their everyday work. By changing their field of possible actions, IE tried to shift the balance of power in the whole web community to create a bias toward Microsoft.

The most widely publicized issue is that IE repeatedly broke many open web standards of the World Wide Web Consortium, and introduced Microsoft's own *de facto* standards and their corresponding technologies in place of them. Given the large market share of Explorer, web developers soon found themselves in a position where they had to decide between optimizing their websites solely for Explorer or devoting extra resources to support open standards-compliant browsers. Because most web developers opted for the first alternative, open standards-compliant browsers had to come up with workarounds to be able to view IE-optimized websites. In this way, they became followers of Microsoft's *de facto* standards. Since Microsoft's *de facto* standards are not bound by constraints of industry-wide consensus, they could be biased toward certain Microsoft technologies without any possibility of compelling and efficient criticism. They can be changed, replaced or revoked arbitrarily, without any discussion or warning, which puts followers in a subordinate position.

For example, Microsoft's ActiveX technology enables content providers to publish dynamic content on their web pages. But ActiveX components – in contrast to standards-compliant JAVA applets – can only be run under the Microsoft Windows operating system. It is extremely hard to port ActiveX technology onto other systems, because both the server components and the necessary Microsoft client libraries are closed source. Even if other system builders would come up with a way to decipher ActiveX technology to emulate the necessary Windows behavior, Microsoft can decide to change its technology arbitrarily (e. g., in a new Windows or IE version), thereby making such efforts futile.

In this way, Microsoft could *leverage* IE's popularity to influence the technological choice of third-party content providers, and then further leverage this choice to *reduce the choices* (the relevant possibilities of action) of its users between Windows and other operating systems. Supposedly, at the final stage of this process, if the user wants to view the dynamic content on the particular site, he will have to use Windows along with Internet Explorer. The operating system, which had traditionally nothing

to do with the rendering of web pages, becomes an "obligatory passage point" (Latour, 1987: 141), through which all users of the website have to pass.

It is this context, in which the self-disciplined adherence to standards could acquire the meaning of "freedom". Mozilla developers build on open World Wide Web Consortium standards instead of the unilateral dictates of an industry giant, so they are less likely to be biased toward particular technologies and they open up a wider field of possible actions for their future users.

6.7.5. Project management – a detailed analysis

Mozilla Firefox made its first appearance under the name "Phoenix" with version 0.1 at 23/09/2002. It was the result of an experimental fork of the Mozilla source, initiated by Dave Hyatt. The Mozilla Firefox Development Charter summarizes the main principles of the project:

> - Delivering the right set of features - not too many or too few (the goal is to create a useful browser, not a minimal browser)
> - Making as few compromises as possible where user experience is concerned. We will not compromise the main line UI to placate an element of the community. Usability is a large area consisting not only of the things one typically considers related to the user experience such as the design of dialog boxes and windows, but also things such as interaction design (looking at how users try and accomplish a task, noting the paths they take and attempting to optimize those paths) and performance (reaction speed from a piece of software is important so as not to annoy the user - perceived speed is often more important than actual speed).
> - Develop and maintain an extension system to allow for research into new areas without affecting the core and to allow for techies, early adopters, web developers and other specific communities to customize their browsers to suit their specific needs without affecting usability or download size for the mass market.
> - Retaining a tight command and control hierarchy. UI design is not a committee driven process. Application design must be nimble and testing is better than discussion, so:
> - Make changes quickly and then get them to people so that we can refine them based on observation of user interactions. (Mozilla Firefox Development Charter)

These rationales reflect a departure from the mistakes of the Netscape/Mozilla development. While Mozilla.org tried to be a complete suite of tools, it was never really good in any particular field. Development efforts splintered and dissipated in the struggle to keep up with the fierce competition in all fronts. Phoenix was conceived to be minimalist in comparison. The "few compromises" clause most probably refers to the ill-fated AOL/Netscape interface issue. In the charter, special emphasis is being laid on usability, but its meaning is left ambivalent: on the one hand, UI design seems to be under the control of a tight hierarchy, while the role of user testing and feedback is also emphasized – along with usability lab tests. Knowing that in the early phase of development, the core Phoenix development was open for invited members only, and that for a long time they did not have access to a usability lab, we can assume that the members of the hierarchy were responsible for user interface decisions. The quick development cycle mentioned in the last point refers to the *nightly builds*, which meant a specific constraint on the project integrity. The automatic building system produced a fully functional, testable version of the browser each day, so the code in the main development line always had to be in a fully consistent state.

Maintaining consistency and "conceptual integrity" (Brooks, 1995) in a project staffed by volunteers on the payroll of different organizations requires special organization and leadership. Contributors do not carry out a centralized "will", and do not take explicit management orders, but their field of possible actions is subject to many constraints.

On the technical level, software tools are used to check and enforce consistency. Mozilla.org contributors have developed *Bonsai* to make it easier for everyone to see whether the source code is in a consistent state, and to find who is responsible for breaking the consistency. They utilize a decent source code versioning system (CVS), supplemented with fine-grained conventions on the processes of submitting changes, working with various development branches, assigning informal rights and so on. They use their own bug tracking system (Bugzilla) to share and follow bug reports.

On the social level, Firefox command-and-control has a more or less decentralized, redundant, flat hierarchy, which reflects the organization of the source code. At the start of each development phase, a special focus group consisting of Mozilla core staff and module owners sets the development schedule, weights milestones, assigns modules, and delegates special tasks (like designing UI concepts), finally creates and publishes a requirements document on the website. The progress is reviewed weekly.

Development in each functional module is led by one or two "module owners" and 1-3 "peers". Their work does not consist of day-to-day management and assigning tasks, their "subordinates" are relatively autonomous. Almost any volunteer can submit bug reports, patches, and code changes in the module on his own. Their most important role – apart from developing new features themselves – is to maintain the conceptual integrity of the module by taking part in the discussions about the suggested changes, reviewing them before they are submitted, and adding improvements to the suggested solutions.

Normally, there are two levels of review. The so-called "super-review" actually precedes the previously mentioned module owner/peer review. "Super" means that this review is aimed at maintaining the overarching project integrity: coding standards, best practices, formal and informal conventions. It also serves to strengthen inter-module consistency and compatibility with future development plans. This is carried out by senior developers of the core group.

Before each grand public release, a new level of review comes in, which is done by so-called *drivers*. Their job is to screen bugs and patches before they are sent to review in order to decide which are issues to which developers/reviewers should devote their precious time before the release deadline, and which are those that can wait until the next release. They must make sure that fixes do not introduce new functionality with unforeseeable side effects, because that might cause a domino effect of problems, and unpredictable delays of the deadline. They also pay close attention to bug reports so that they don't get forgotten, and that all associated changes get consistently carried out (patches are applied to all development branches, etc.). In the Mozilla hierarchy, it is the drivers' work what most closely resembles classic management – prioritizing tasks to optimize workflow on a daily basis – but in contrast to managers, they do not give out orders to the participants.

All through the development process, power is exerted discreetly through the fields of the Bugzilla database. The classification of change requests and bug reports is an important power device. Each bug and change request is assigned to a bug owner (often corresponding to the module owner or a driver), who assigns a *priority* to the bug. The priority determines the chance that the bug will actually be considered (the *WONTFIX* or *INVALID* categorization means the rejection of the bug report). There are also conventions on the title and the content of the change request. If one breaks these conventions, it might simply result in ignoring of the request, but in severe cases, it might result in revoking access to the Bugzilla database. The +/– flags resulting from the reviews are also documented in the Bugzilla. The final barrier to pass in order to carry out the necessary modifications is constituted by the access control of CVS system, and the conventions, which govern its use.

These layers constitute an explicit *entry barrier* on the admissible user contributions. Should the user try to bypass them, her request will be ignored. Particularly, new feature and interface change requests from unauthorized users will most probably be ignored, especially before releases.

Beside these explicit barriers, there are also *disciplinary* power devices in play. Many people have rights to submit changes to the CVS, but if it turns out during the nightly build that the change breaks the consistency of the code, the "tree goes red" on the *Bonsai* display, and a link will be displayed with the name of the developer who is to be blamed. This is considered a big shame and everyone is expected to avoid it.

Having seen the detailed mechanism of the Mozilla development, we might notice that these processes are indeed powerful *strategies*, almost indistinguishable from those of commercial projects. They are indispensable means of maintaining the integrity of the project and the code as the *proper place* of the community, and it is striking how people – even volunteers – subject themselves to them. Of

course, these strategies can be circumvented if the user builds an own extension module or a private patch – but then it won't make it into the main development line, and it won't get the attention and improvements of Mozilla contributors.

6.8. Conclusions

I carried out my analysis by reflecting upon the fields of possible action of users and developers in various episodes of the history of the Mozilla Firefox project. I reconstructed the changes and shifts these fields underwent, and as they were perceived by the actors.

Finally, I bring into attention that the field of possible actions is not independent from the analyst's or the actor's perspective. It is a *perceived* space, arising from a*n interpretation of the situation as a space of open alternatives*, which widens as our horizon of conceivable courses of action is extended by contemplating new alternatives or by turning highly implausible alternatives into plausible ones; and it narrows, as possibilities of action are judged improbable or irrelevant. As our examples suggest, the existence of such spaces depend greatly on *how consistently novel interpretations can be maintained and conventionalized in the social space in which the technical artifact is embedded.* Tactical reinterpretations can turn into strategies, if enough people align themselves and act according to them. As Foucault's Panopticon-example suggests, the power field exists even if no action is carried out, but the participants themselves perceive their possibility and interiorize this perception as reality.

7. Bibliography

7.1. Sources relevant for the Ada case

7.1.1. Comprehensive archive of Ada-related documents

http://www.iste.uni-stuttgart.de/ps/AdaBasis/pal_1195/ada/ajpo/

7.1.2. Websites dedicated to Ada-related activity

http://www.acm.org/sigada/
http://www.adahome.com/
http://www.adaic.com/
http://www.ada-europe.org/

7.1.3. FAQ about the closure of the Ada Joint Program Office

http://sw-eng.falls-church.va.us/ajpofaq.html

7.1.4. Edsger W. Dijkstra Archive

http://www.cs.utexas.edu/users/EWD/transcriptions/transcriptions.html

7.1.5. History of Ada – primary documents and contemporary reflections

Ada Information Clearinghouse. 1991 "Overview of U.S. Air Force Report - Ada and C++: A Business Case Analysis." http://archive.adaic.com/docs/flyers/83cplus.html.

Carlson, W. E., Druffel, L. E., Fisher, D. A., Whitaker W. A. 1980 "Introducing Ada" Proceedings of the ACM 1980 annual conference ACM '80, 263-271

Cohen, P. M. 1981 "From HOLWG to AJPO: Ada in transition" ACM SIGAda Ada Letters 1(1): 22-25

[CPPCUADoD] Committee on the Past and for the Use of Ada in the Department of Defense. 1997 Ada and Beyond, Software Policies for the Department of Defense. Washington, D. C.: National Academy Press.

daCosta, Robert. 1984 (March) "The History of Ada." Defense Science Magazine.

Davis, N. C. 1978 "The Soviet Bloc's Unified System of Computers" Computing Surveys 10(2): 93-122

DeRemer, F., Kron, H. 1975 "Programming-in-the large versus programming-in-the-small" ACM SIGPLAN Notices , Proceedings of the international conference on Reliable software 10(6): 114-121

Fisher D. A. 1975 (Aug) Woodenman Set of Criteria and Needed Characteristics for a Common DoD High Order Programming Language", Institute for Defense Analyses Working Paper

Fisher D. A. 1978 “DoD's common programming language effort” Computer 11(3): 24-33

Ichbiah, J. D. 1979 (June) “Preliminary Ada reference manual” ACM SIGPLAN Notices 14(6): 1-145
Ichbiah, J. D. 1979 (June) “Rationale for the Design of the ADA Programming Language” ACM SIGPLAN Notices 14(6): 1-145
Ichbiah, J. D. 1984 “Ada: Past, Present, Future An Interview with Jean Ichbiah, the Principal Designer of Ada” Communications of the ACM 27(10): 990-997

Ichbiah, J. D., Barnes, G. P. B., Firth, R. J., Woodger, M. 1983 [RATL] Rationale for the Design of the Ada Programming Language, HONEYWELL Systems and Research Center

Lieblein, E. 1986 “The Department of Defense software initiative – a status report” Communications of the ACM 29(8): 734-744

Licklider, J. C. R. 1960 (March) IRE Transactions on Human Factors in Electronics, volume HFE-1: 4-11

Lions, J. L. 1996 (July 19) ARIANE 5 Flight 501 Failure – Report by the Inquiry Board

Paige, E.J. 1997 „Use of the Ada Programming Language.” Assistant Secretary of Defense Memorandum. Apr. 29. http://www.adahome.com/articles/1997-04/po_memopaige.html

Software Productivity Consortium. 1983 [AQS] Ada Quality and Style: Guidelines for Professional Programmers. Van Nostrand Reinhold.

U. S. Department of Defense. 1977 (Jul) “Revised Ironman Requirements for High Order Computer Programming Languages”
U. S. Department of Defense. 1978 (June) “Requirement for High Order Computer Programming Languages. STEELMAN”
U. S. Department of Defense. 1987 ”Computer programming Language Policy.” Department of Defense Directive 3405.1

Whitaker, W.A. 1993 „Ada - The Project, The DoD High Order Language Working Group”. ACM SIGPLAN Notices 28: 3.

7.1.6. Criticism and Assessment

Baker, H. G. 1997 “I have a feeling we're not in emerald city anymore” ACM SIGPLAN Notices 32(4) 22-26

Bennett, D. A., Kornman, B. D., Wilson J. R. 1982 “Hidden costs in Ada” ACM SIGAda Ada Letters 1(4): 9-20.

Borning, Alan. 1987 “Computer System Reliability and Nuclear War” Communications of the ACM 30(2): 112-131

Chelini, J. V. 1987 “The impact of the Ada language on resource allocation, programmer productivity, and project performance” Proceedings of the Joint Ada conference fifth national conference on Ada technology and fourth Washington Ada Symposium, 183-186

Dijkstra, E. W. 1975 [EWD514] “On a language proposal for the Department of Defence.”
Dijkstra, E. W. 1975 [EWD526] “Comments of "Woodenman" HOL Requirements for the DoD.”
Dijkstra, E. W. 1978 [EWD658] “On language constraints enforceable by translators. An open letter to Lt. Col. William A. Whitaker”
Dijkstra, E. W. 1978 [EWD659] “On the BLUE Language submitted to the DoD.”
Dijkstra, E. W. 1978 [EWD660] “On the GREEN Language submitted to the DoD.”
Dijkstra, E. W. 1978 [EWD661] “On the RED Language submitted to the DoD.”
Dijkstra, E. W. 1978 [EWD662] “On the YELLOW Language submitted to the DoD.”
Dijkstra, E. W. 1978 [EWD663] “DOD-1: The Summing Up”

Feinberg, D.A. 1987 „Non-Technical Aspects of Using Ada.” Pp. 180-182 in Proceedings of the Joint Ada conference fifth national conference on Ada technology and fourth Washington Ada Symposium. Arlington: George Washington University.

Gerhardt, M. S. 1989 “The real transition problem or don't blame Ada” Proceedings of the conference on TRI-Ada '88. Pp. 620-645.

Parnas, D. L. 1985 “Software Aspects of Strategic Defense Systems” Communications of the ACM 28(12): 1326-1335

Reifer, D.J. 1987 “Ada's impact: a quantitative assessment” Proceedings of the 1987 annual ACM SIGAda international conference on Ada, 1-13
Reifer, D.J. 1996 ”Quantifying the Debate: Ada vs. C++.” STSC CrossTalk (Jul)
http://www.stsc.hill.af.mil/crosstalk/1996/07/quantify.asp.

Smith, D. A. 1987 “Mechanisms for abstraction in Ada” Proceedings of the Joint Ada conference fifth national conference on Ada technology and fourth Washington Ada Symposium WADAS '87, 126-132

Kerner, J. 1992 “Where is Ada headed? (panel): the facts behind the myths.” Pp. 563 – 568 in Proceedings of the conference on TRI-Ada 92.

Kling, R. 1974 “Computers and social power” ACM SIGCAS Computers and Society 5(3): 6-11
Kling, R. 1978 “Automated Welfare Client-Tracking and Service Integration: The Political Economy of Computing”Communications of the ACM 21(6): 484-493

Kling, R. and Scacchi, W. 1979 “The DoD common high order programming language effort (DoD-1): what will the impacts be?” ACM SIGPLAN Notices 14(2): 29-43

Tang, L. S. 1992 “A comparison of Ada and C++” Proceedings of the conference on TRI-Ada '92 338-349.

Wichmann, B.A. 1984 “Is Ada too big? A designer answers the critics” Communications of the ACM 27(2): 98-103

Wheeler, D. A. 1997 "Ada, C, C++, and Java vs. the Steelman" ACM SIGAda Ada Letters 17(4): 88-112

7.2. Sources relevant for the Python case

7.2.1. Policies, Processes and Guidelines

Repository of Python Enhancement Proposals
http://www.python.org/dev/peps/

(PEP 1): "PEP Purpose and Guidelines"
http://www.python.org/dev/peps/pep-0001/
(PEP 5): "Guidelines for Language Evolution"
http://www.python.org/dev/peps/pep-0005/
(PEP 20): "The Zen of Python"
http://www.python.org/dev/peps/pep-0020/
(PEP 42): "Feature Requests"
http://www.python.org/dev/peps/pep-0042/
(PEP 8): "Style Guide for Python Code"
http://www.python.org/dev/peps/pep-0008/
(PEP 257): "Docstring Conventions"
http://www.python.org/dev/peps/pep-0257/
(PEP 3099): "Things that will Not Change in Python 3000"
http://www.python.org/dev/peps/pep-3099/

(PDP) Python's Development Process
http://www.python.org/dev/process/

Python Culture
http://www.python.org/dev/culture/

7.2.2. Criticism and Assessment

Arnold, Ken. 2004 "Style is Substance".
http://www.artima.com/weblogs/viewpost.jsp?thread=74230 (Accessed: jan. 31, 2007).

Bicking, I. 1995 "UnZen of UnPython."
http://blog.ianbicking.org/unzen-of-unpython.html (Accessed dec. 1, 2006).

Rossum, G. v. 1996, "Foreword" in Mark Lutz, Programming Python (1st ed.), O'Reilly & Associates
http://www.python.org/doc/essays/foreword.html
Rossum, G. v. 2001, "Foreword" in Mark Lutz, Programming Python (2nd ed.), O'Reilly & Associates
http://www.python.org/doc/essays/foreword2.html
Rossum, G. v. 2006 "Language design is not just solving puzzles."
http://www.artima.com/weblogs/viewpost.jsp?thread=147358.

7.2.3. Mailing list archives

The Python-Dev Archives
http://mail.python.org/pipermail/python-dev/

The Python-list Archives
http://mail.python.org/pipermail/python-list/

comp.lang.python
http://groups.google.com/group/comp.lang.python

7.2.4. Turbogears

Main webpage
http://www.turbogears.org/

MoinMoin (Wiki)-based documentation
http://docs.turbogears.org/

The TurboGears Trac
http://trac.turbogears.org/

TurboGears Discussion Group
http://groups.google.com/group/turbogears

7.3. Mozilla Firefox

7.3.1. History of Mozilla Firefox

Browser Market Share for February, 2007
http://marketshare.hitslink.com/report.aspx?qprid=0

Eich , Brendan 26/10/1998. Mozilla Development Roadmap
http://www.mozilla.org/roadmap/roadmap-26-Oct-1998.html

Eich, Brendan. 15/05/1998. Mozilla Stabilization Schedule (Was: Launch date of Netscape Communicator 5.0?)
http://groups.google.com/group/netscape.public.mozilla.general/msg/60b3fb3f85ad34c0

Elliott, Rick. 21/05/1998. Portions of the UI are expressable as HTML.
http://groups.google.com/group/netscape.public.mozilla.general/browse_thread/thread/ca64498041386c17/a6f6ef1638f09f6d?lnk=st&q=group%3Anetscape.public.*+raptor&rnum=21#a6f6ef1638f09f6d

Hickman, Kipp E. B. 15/04/1998. Announcing Raptor.

http://groups.google.com/group/netscape.public.mozilla.general/browse_thread/thread/7e6cea5bc8a090a6/fd3f79342cd54810?lnk=st&q=group%3Anetscape.public.*+raptor&rnum=1#fd3f79342cd54810

History of Mozilla Firefox
http://en.wikipedia.org/wiki/History_of_Mozilla_Firefox

History of Mozilla Application Suite
http://en.wikipedia.org/wiki/History_of_Mozilla_Application_Suite

Gessner, Rick and Angus Davis. 25/09/98. A Great Week for NGLayout!
http://groups.google.com/group/netscape.public.mozilla.layout/browse_thread/thread/f46e76201c3e96be/e5666e51b93c1668?lnk=st&q=group%3Anetscape.public.*+nglayout&rnum=1#e5666e51b93c1668

Goodger, Ben. 10/7/2004. Firefox 1.0 Roadmap
http://www.mozilla.org/projects/firefox/roadmap-1.0.html

Goodger, Ben. 06/02/2006. Where Did Firefox Come From?
http://weblogs.mozillazine.org/ben/archives/009698.html

Junnarkar, Sandeep and Tim Clark. 24/11/1998. AOL buys Netscape for $4.2 billion
http://news.com.com/2100-1023-218360.html

Knauss, Greg. 10/04/2000. Skin Cancer.
http://www.suck.com/daily/2000/04/10/daily.html

Netscape
http://en.wikipedia.org/wiki/Netscape_Communications_Corporation

Spolsky, Joel. 06/04/2000. Things You Should Never Do, Part I
http://www.joelonsoftware.com/articles/fog0000000069.html

Spolsky, Joel. 20/11/2000. Netscape Goes Bonkers
http://www.joelonsoftware.com/articles/fog0000000027.html

Trudelle, Peter. XPToolkit Goals. 2/9/99(1)
http://www.mozilla.org/xpfe/goals.html

Trudelle, Peter. XPToolkit Executive Review. 2/9/99(2)
http://www.mozilla.org/xpfe/ExecReview.html

Turnbull, Andrew. 21/02/2006. A Visual Browser History, from Netscape 4 to Mozilla Firefox
http://community.wvu.edu/~ast002/mozilla/history.html

Zawinski, Jamie. 1999. Resignation and postmortem.
http://www.jwz.org/gruntle/nomo.html

7.3.2. Debates

Netscape lost its edge? 24/03/1999.
http://groups.google.com/group/netscape.public.mozilla.general/browse_thread/thread/e96ea44eda28dd4b/91c6ce0783905711#91c6ce0783905711

"Out of the Box" Mozilla? 16/06/1999.
http://groups.google.com/group/netscape.public.mozilla.general/browse_thread/thread/e3324a9e5f483723/f2fb0de30373e254#f2fb0de30373e254

Netscape == Mozilla. 24/08/1999.
http://groups.google.com/group/netscape.public.mozilla.embedding/browse_thread/thread/957f58182ad18995/0b3f4b312f2e8f43#0b3f4b312f2e8f43

New Skins coming to Mozilla. 22/10/1999.
http://groups.google.com/group/netscape.public.mozilla.xpfe/browse_thread/thread/45c05d613d64ae12/c73a9d47737b61de#c73a9d47737b61de

UI spoofing attacks. 28/10/1999.
http://groups.google.com/group/netscape.public.mozilla.xpfe/browse_thread/thread/b4cc81c957405278/5f601725a5acdb34#5f601725a5acdb34

NUI. 11/12/1999.
http://groups.google.com/group/netscape.public.mozilla.ui/browse_thread/thread/608dfb4a8619841d/1e5a97a0cc564138#1e5a97a0cc564138

The Navigator Window UI. 16/12/1999.
http://groups.google.com/group/netscape.public.mozilla.xpfe/browse_thread/thread/3fd32ba4a9ea2d9b/22fa17c556061c1b#22fa17c556061c1b

Veditz, Dan. 02/02/2000 Mozilla XPInstall capabilities
http://groups.google.com/group/netscape.public.mozilla.general/browse_thread/thread/39c64bdc102de60/aa815e93916cec71#aa815e93916cec71

Thomas, Matthew. 06/03/2000. The whys and wherefores of sidebar tabs.
http://groups.google.com/group/netscape.public.mozilla.ui/browse_thread/thread/4199d301c7b34109/ad93664c2cfec6fc#ad93664c2cfec6fc

Toolbar buttons have a horrible grey bit at the bottom. 27/03/2000.
http://groups.google.com/group/netscape.public.mozilla.ui/browse_thread/thread/d728acdde8b37e8b/9725a098d7bdb3a5#9725a098d7bdb3a5

Major site may no longer support NS. 12/04/2000.
http://groups.google.com/group/netscape.public.mozilla.general/browse_thread/thread/d4e91501a14731a8/80e34348957b57e9#80e34348957b57e9

Content-to-Chrome Ratio. 20/04/2000.

http://groups.google.com/group/netscape.public.mozilla.ui/browse_thread/thread/9e8598502b0f6b35/d2cd3cd75f5dd6f0#d2cd3cd75f5dd6f0

Thomas, Matthew. 26/04/2000. Native windows
http://groups.google.com/group/netscape.public.mozilla.ui/browse_thread/thread/39c52e38896b2f9c/8089ae9a6b047a33#8089ae9a6b047a33

Should Mozilla Have it's Own Skin? 21/06/2000.
http://groups.google.com/group/netscape.public.mozilla.ui/browse_thread/thread/84b118ef333e4d33/567d04c792f6e606#567d04c792f6e606

Usability Studies at Netscape. 05/08/2000.
http://groups.google.com/group/netscape.public.mozilla.ui/browse_thread/thread/5486164195b8e533/8beba3bfa96af8ad#8beba3bfa96af8ad

Bug 48205 – Use Classic Skin as default. 09/08/2000.
https://bugzilla.mozilla.org/show_bug.cgi?id=48205

Modern Skin Development Update. 22/08/2000.
http://groups.google.com/group/netscape.public.mozilla.xpfe/browse_thread/thread/b272f325b7eb5657/f1e71168073be0fc#f1e71168073be0fc

MacInTouch Reader Reports: Netscape 6. 14/11/2000.
http://www.macintouch.com/netscape6.html
http://www.macintouch.com/netscape6part2.html

Salon on Mozilla. 12/03/2002
http://www.mozillazine.org/talkback.html?article=2167

More Mozilla RC1 Reviews. 25/04/2002
http://www.mozillazine.org/talkback.html?article=2196

Veditz, Daniel. 22/06/2002. Mozilla 1.0.1 update...
http://groups.google.com/group/netscape.public.mozilla.seamonkey/msg/ab69e3532f258daa?q=ui+fork

Enough. 24/09/2002.
http://groups.google.com/group/netscape.public.mozilla.general/browse_thread/thread/794b732cc89cca18/b37ac71521449a64#b37ac71521449a64

AOL Cuts Remaining Mozilla Hackers. 15/07/2003.
http://www.mozillazine.org/talkback.html?article=3422

7.3.3. Policy documents

Bugzilla@Mozilla – Bug Writing Guidelines
https://bugzilla.mozilla.org/page.cgi?id=bug-writing.html

Frequently Asked Questions about mozilla.org's Code Review Process
http://www.mozilla.org/hacking/code-review-faq.html

Goodger, Ben. 11/28/2004. Mozilla Firefox Development Charter
http://www.mozilla.org/projects/firefox/charter.html

Module Owners
http://www.mozilla.org/owners.html

mozilla.org Staff
http://www.mozilla.org/about/staff

mozilla.org Drivers
http://www.mozilla.org/about/drivers

Review rules
http://www.mozilla.org/projects/firefox/review.html

7.3.4. Personal blogs

Goodger, Ben. 01.07.2006. Creativity
http://weblogs.mozillazine.org/ben/archives/016665.html

Goodger, Ben. 05.05.2006. The "Joy" of XUL?
http://weblogs.mozillazine.org/ben/archives/015929.html

7.3.5. Other resources

The Extensions Mirror
http://www.extensionsmirror.nl/

Firefox Add-ons
https://addons.mozilla.org/en-US/firefox/browse/type:1

Developer Forums
http://www.mozilla.org/community/developer-forums.html

Mailing lists
https://lists.mozilla.org/listinfo

7.3.6. Debian-Mozilla affair

Debian Bug report #354622 Uses Mozilla Firefox trademark without permission
http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=354622

Trademark Clarification
http://weblogs.mozillazine.org/ben/archives/017159.html

IceWeasel

http://en.wikipedia.org/wiki/Iceweasel

7.4. Sources on software methodology

Beck, K. 1999 *Extreme programming explained: Embrace change.* Addison-Wesley Professional.

Karlsson, Björn. 2005. *Beyond the C++ Standard Library: An Introduction to Boost.* Addison Wesley Professional

Boehm, B. 1979 "Software Engineering As It Is." *Proceedings of the 4th international conference on Software engineering.* Munich, Germany. 11-21.

Boehm, B. & Turner, R. 2004 *Balancing Agility and Discipline.* Addison-Wesley.

Brooks, F. P. Jr. 1987 "No silver bullet: essence and accidents of software engineering". *Computer* 20(4): 10-19
Brooks, F. P. Jr. 1995 *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition.* Reading, MA: Addison-Wesley.

DeMarco, T. & Lister, T. 1999 *Peopleware (2nd ed.)* New York: Dorset House Publishing.

Fogel, Karl. 1995. *Producing Open Source Software - How to Run a Successful Free Software Project.* O'Reilly.

Hunt, A. & Thomas, D. 1999 *The Pragmatic Programmer: From Journeyman to Master.* Addison-Wesley.

McBreen, P. 2002. *Software Craftsmanship - The New Imperative.* Addison - Wesley.

7.5. Philosophy

7.5.1. Hermeneutics and Philosophy of Technology

Binzberger, Viktor. 2006. "Mi teszi Heideggert relevánssá a Mesterséges Intelligencia-kutatás filozófiai kritikái számára?" [What makes Heidegger relevant for the philosophical critiques of Artificial Intelligence?] Pp. 209-230 In: Fehér, Márta, Zemplén, Gábor, Binzberger, Viktor (eds). 2006. *Értelem és történelem (Understanding and History), Tudománytörténeti és Tudományfilozófiai évkönyv*, II. kötet. Budapest: L'Harmattan.

Binzberger, Viktor. 2006b. "A szoftverhiba jelensége hermeneutikai megközelítésből" [The phenomenon of the software bug from a hermeneutical standpoint.] *Világosság*, 2006(3): 27-34.

Brighham, M. & Introna, L. 2006. "Hospitality, improvisation and Gestell: a phenomenology of mobile information." *Information Technology & People* 21: 140-153

Capurro, R. 1992. "Informatics and Hermeneutics." Pp. 363-375 in Floyd, Züllighoven, Budde & Keil-Slawik (eds.) *Software Development and Reality Construction.* Berlin, Heidelberg, New York: Springer-Verlag.

Ciborra, C. U. 1998. "From tool to Gestell – Agendas for managing the information infrastructure." *Information Technology & People* 11(4): 305-327.

Dreyfus, H. L. & Spinoza, C. 1997. „Highway Bridges and Feasts: Heidegger and Borgmann on How to Affirm Technology." After Post-Modernism Conference website.
http://www.focusing.org/ dreyfus.html (Accessed dec. 1, 2005).

Dreyfus, H. L. 1991. *Being-in-the-World.* Massachusetts Institute of Technology.
Dreyfus, H. L. 1994. *What Computers Still Can't Do: A critique of artificial reason.* Cambridge, Massachusetts: The MIT Press.
Dreyfus, H. L. 1998. „Why we do not have to worry about speaking the language of the computer." *Information Technology & People* 11(4): 281-289.

Gregory, Wanda Torres. 1998. "Heidegger On Traditional Language And Technological Language." Presentation on the 20th World Congress on Philosophy.
Gregory, Wanda Torres. 2001. "Heidegger, Carnap, and Quine at the Crossroads of Language". Current Studies in Pehnomenology and Hermeneutics, 1(NA).

Heelan, P. A. 1989. „Yes! There Is a Hermeneutics of Natural Science: A Rejoinder to Markus", *Science in Context* 3(2): 477-488.
Heelan, P. A. 1997. "After Post-Modernism: The Scope of Hermeneutics in Natural Science." After Post-Modernism Conference website.
http://www.focusing.org/apm_papers/heelan.html (Accessed dec. 1, 2006)
Heelan, Patrick A. 2005. „Carnap and Heidegger: Parting Ways in the Philosophy of Science".
http://www.georgetown.edu/faculty/heelanp/Heid.Carnap.02.htm (Accessed dec. 1, 2006)

Heelan, P.A. & Schulkin. 2003. "Hermeneutical Philosophy and Pragmatism: A Philosophy of Science." pp. 138-154. In: J. Scharff, R.C. & Dusek, V. (eds.) *Philosophy of Technology: An An-thology.* Oxford: Blackwell Publishing.

Heidegger, M. 1947. *Platons Lehre von der Wahrheit, mit einem Brief über den "Humanismus".* Bern: Francke.
Heidegger, M. 1950. "Das Ding." in *GA7.* Vittorio Klostermann.
Heidegger, M. 1957. *Hebel - Der Hausfreund.* Pfullingen: Günther Neske.
Heidegger, M. 1962. *Die Technik und die Kehre.* Pfullingen: Günther Neske.
Heidegger, M. 1971. "Building, Dwelling, Thinking." In: *Poetry, Language, Thought.* (trans. Albert Hofstadter) New York: Harper Colophon Books.
Heidegger, M. 1977. "The Age of the World Picture" In: *The Question Concerning Technology and Other Essays* (trans. William Lovitt). New York: Harper & Row
Heidegger, M. 1977. "The Question Concerning Technology" In: *The Question Concerning Technology and Other Essays* (trans. William Lovitt). New York: Harper & Row
Heidegger, M. 1996. [BT] *Being and Time: A Translation of Sein und Zeit* (Traslated by John Stambaugh). New York: State University of New York Press.
Heidegger, M. 1998. "Letter on Humanism". In: *Pathmarks* Cambridge & New York: Cambridge University Press. pp. 250–1.
Heidegger, M. 2000. *Introduction to Metaphysics.* Yale University Press.
Heidegger, M. 2002. "Das Ende der Philosophie und die Aufgabe des Denkens. [Vortrag]" in *GA14.* Vittorio Klostermann.

Heidegger, M. 2002. "Der Weg zur Sprache. [Vortrag München, Berlin]" in *GA12*. Vittorio Klostermann.
Heidegger, M. 2003. [SZ] *Sein und Zeit (GA2)*. Vittorio Klostermann.

Heim, M. 1993. *The Metaphysics of Virtual Reality*. New York: Oxford University Press.

Ihde, Don. 1990. "Program One. A Phenomenology of Technics." Pp. 72-108 in *Technology and the Lifeworld: From Garden to Eden*. Bloomington: Indiana University Press.
Ihde, Don. 1999. *Expanding Hermeneutics: Visualism in Science*. Northwestern University Press.
Ihde, Don. 2003. "Heidegger's Philosophy of Technology." In: Scharff, Dusek (eds.) *Philosophy of Technology: An Anthology*. Oxford: Blackwell Publishing.
Ihde, Don. 2003. "A Phenomenology of Technics." In: Scharff, R.C. & Dusek, V. (eds.) *Philosophy of Technology: An Anthology*. Oxford: Blackwell Publishing

Kisiel, T.J. 2001. "Heidegger és az új tudománykép." In: Margitay T. & Schwendtner T. (eds.), *Hermeneutika és a természettudományok*. Budapest: Áron kiadó

Kochan, J. 2005. *A Poetics of Tool-use - Explorations in Heidegger and Science Studies*. PhD thesis, Churchill College.

Márkus, G. 1987. „Why is There No Hermeneutics of Natural Science? Some Preliminary Theses." *Science in Context* 1: 5-51.

Mitcham, Carl. 1994. *Thinking Through Technology. The Path Between Engineering and Philosophy*. Chicago and London: The University of Chicago Press.
Mitcham, Carl. 2004. "Introduction." In: Floridi, L. (ed.) *The Blackwell Guide to the Philosophy of Computing and Information*. Oxford: Blackwell Publishing

Mohanty, J. N. 1984. "Transcendental Philosophy and the Hermeneutic Critique of Consciousness." In: Shapiro, G., Sica, A. (eds.) *Hermeneutics: Questions and Prospects*. Amherst.

Ramberg, B. and Gjesdal, K. 2005. *Hermeneutics*. Entry in the Stanford Encyclopedia of Philosophy http://plato.stanford.edu/entries/hermeneutics/

Schwendtner, Tibor. 2000. *Heidegger tudományfelfogása*. Budapest: Osiris.

Schwendtner, T.; Ropolyi, L., Kiss, O. (eds.). 2001. *Hermeneutika és a természettudományok* (Hermeneutics and the Natural Sciences). Budapest: Áron kiadó.

Shapiro, G., Sica, A. 1984. *Hermeneutics: Questions and Prospects*. Amherst.

Scharff, R. C. & Dusek, V. (eds.) 2003. *Philosophy of Technology: An Anthology*. Blackwell Publishing.

Verbeek, P.-P. 2005. *What Things Do*. (Trans. Robert P. Crease) Pennsylvania: The Pennsylvania State University Press.

Winograd, T. & Flores, F. 1987. *Understanding Computers and Cognition*. Norwood, NJ: Ablex Corporation.

7.5.2. Power

Dreyfus, H. 1984. "Beyond Hermeneutics: Interpretation in Late Heidegger" In: Shapiro, G., Sica, A. (eds.) *Hermeneutics: Questions and Prospects*. Amherst.
Dreyfus, H. 2001. "Being and Power: Heidegger and Foucault," *International Journal of. Philosophical Studies* 4(1): 1-48.

de Certeau, Michel. 1984. *The Practice of Everyday Life*. Translated by Steven F. Rendail. Berkeley: University of California Press

Feenberg, A. 1991. *Critical Theory of Technology*. New York: Oxford University Press.
Feenberg, A. 1995. "Subversive Rationalization: Technology, Power, and Democracy" in Feenberg & Hannay (eds.) *Technology and the Politics of Knowledge*. Bloomington and Indianapolis: Indiana University Press.
Feenberg, A. 1996. „Marcuse or Habermas: Two Critiques of Technology." *Inquiry* 39: 45-70.
Feenberg, A. 1999. *Questioning Technology*. New York: Routledge.
Feenberg, A. 2000. "From Essentialism to Constructivism: Philosophy of Technology at the Crossroads." Pp. 294-315 in Higgs, Strong, Light (eds.) *Technology and the Good Life*. University of Chicago Press.
Feenberg, A. 2003. "Critical Evaluation of Heidegger and Borgmann." in Scharff & Dusek (eds.) *Philosophy of Technology: An Anthology*. Blackwell Publishing.

Foucault, Michel. 1995. *Discipline and Punish: The Birth of the Prison*. (transl. Alan Sheridan) New York: Vintage Books.
Foucault, Michel. 1980. *Power/Knowledge*. New York: Pantheon.
Foucault, M. 1982 "The subject and power." Pp. 208-226 in Hubert Dreyfus & Paul Rabinow (eds.) *Michel Foucault: Beyond Structuralism and Hermeneutics*. London: Harvester Wheatsheaf.

Habermas, J. 1987. *Theorie des kommunikativen Handelns*. Frankfurt a.M.: Suhrkamp.
Habermas, J. 1987. *Knowledge and Human Interest*. Polity Press.
Habermas, J. 1989. "Technology and Science as »Ideology«" in Seidman (ed.) *Habermas On Society and Politics: A Reader*. Boston: Beacon Press.

Heiskala, Risto. 2001. "Theorizing power: Weber, Parsons, Foucault and neostructuralism." *Social Science Information* 40(2): 241-264

Joerges, B. 1999. "Do Politics Have Artefacts?" *Social Studies of Science* 29(3): 411-431.

Lessig, L. 1999. *Code and Other Laws of Cyberspace*. New York: Basic Books
Lessig, L. 2006. *Code version 2.0*. New York: Basic Books

Mann, M. 1986. *The Sources of Social Power. Volume I: A History of Power from the Beginning to A.D. 1760*. Cambridge: Cambridge University Press.

Parsons, T. 1960. "The Distribution of Power in American Society", in *T. Parsons Structure and Process in Modern Societies*. Pp. 199-225. Glencoe, IL: Free Press.

Pickett, Brent L. 1996. Foucault and the Politics of Resistance. *Communications of the ACM* 28(4): 445-466.

Polányi, M. 1962. "The Republic of Science: Its Political and Economic Theory." *Minerva* 1: 54-74.

Raymond, E. S. 1999. *The Cathedral & the Bazaar*. O'Reilly.
http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/
Raymond, E. S. 2000. November. "Epilog: Netscape Embraces the Bazaar"
http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s13.html

Rouse, Joseph. 1987. *Knowledge and Power: Toward a Political Philosophy of Science*. Cornell University Press.
Rouse, Joseph. 1996. *Engaging Science: How to Understand Its Practices Philosophically*. Cornell University Press.
Rouse, Joseph. 1999. "Understanding Scientific Practices: Cultural Studies of Science as a Philosophical Program" pp. 95-109 in Biagioli (ed.) *The Science Studies Reader*. New York, London: Routledge.

Truscello, M. 2003. "The Architecture of Information: Open Source Software and Tactical Post-structuralist Anarchism." *Postmodern Culture* 13(3)

Weber, Max. 1978. *Economy and Society. An Outline of Interpretative Sociology*, ed. G. Roth and C. Wittich. Berkeley, Los Angeles and London: University of California Press. (Orig. published 1922.)

Winner, L. 1986. "Do Artifacts Have Politics?" in *The Whale and the Reactor*. Chicago: Univ. of Chicago.
Winner, L. 2005. "Upon opening the black box and finding it empty: Social Constructivism and the Philosophy of Technology." In: Scharff, R.C. & Dusek, V. (eds.) *Philosophy of Technology: An Anthology*. Oxford: Blackwell Publishing

7.6. Sociology and History of Technology

Bijker, W., Hughes, T. P. & Pinch, T. 1987 *The Social Construction of Technological Systems: New Directions in the Sociology and History of Technology*. Cambridge, Mass.: MIT Press.

Bloor, D. 1983. *Wittgenstein: A Social Theory of Knowledge*. Macmillan and Columbia.

Collins, H. M. 1990. *Artificial Experts: Social Knowledge and Intelligent Machines*. Cambridge, Massachusetts: The MIT Press.

Christensen, C. M. 1997. *The Innovator's Dilemma*. Boston, Mass.: Harvard Business School Press

Edwards, P. N. 1996. *The Closed World*. The MIT Press.

Galison, Peter. 1996. "Computer Simulations." In: P. Galison and D. Stump (eds.) *The Disunity of Science* pp. 118-157.

Hounshell, D. 1992. "Du Pont and Large-Scale R&D". In: Peter Galison and Bruce Hevly (eds.) *Big Science: The Growth of Large-Scale Research.* Stanford, CA: Stanford University Press.

Latour, B. 1987. *Science in Action.* Cambridge, Mass.: Harvard University Press.
Latour, B. 1992. "Where Are the Missing Masses? The Sociology of a Few Mundane Artifacts" in Bijker, Law (eds.) *Shaping Technology / Building Society: Studies in Sociotechnical Change.* Cambridge, Massachusetts: MIT Press.
Latour, B. 2004. "Why Has Critique Run out of Steam? From Matters of Fact to Matters of Concern." *Critical Inquiry* 30: 225-248

Suchman, L. 1987. *Plans and Situated Actions: The Problem of Human-Machine Communication.* Cambridge, England: Cambridge Univ. Press.
Suchman, L. 1988. "Representing practice in cognitive science." Pp. 301-322 in Michael Lynch & Steve Woolgar (eds.) *Representation in scientific practice.* Kluwer Academic Publishers.

Susi, Tarja. 2006. *The Puzzle Of Social Activity The Significance Of Tools In Cognition And Cooperation.* Linköping Studies in Science and Technology Dissertation No. 1019

Turkle, S. 1999. "What Are We Thinking about When We Are Thinking about Computers?" Pp. 95-109 in Biagioli (ed.) *The Science Studies Reader.* New York, London: Routledge.

7.7. Cognitivist and postcognitivist perspectives

Agre, P. E. 1992. "Formalization as a Social Project." *Quarterly Newsletter of the Laboratory of Comparative Human Cognition* 14: 25-27.
Agre, P. E. 2002. "The Practical Logic of Computer Work." in Matthias Scheutz (ed.) *Computationalism: New Directions.* Cambridge: MIT Press.

Brooks, R.A. 1991. "Intelligence without representation." *Artificial Intelligence* 47: 139–159.

John M. Carroll (ed.). 2003. *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science.* Morgan Kaufmann.

Clark, Andy. 1997. *Being There.* MIT Press.

Dourish, Paul. 2001. *Where The Action Is: The Foundations of Embodied Interaction.* MIT Press.

Floridi, Luciano (ed.). 2004. *The Blackwell Guide to the Philosophy of Computing and Information.* Oxford: Blackwell Publishing

Halverson, C. 1994, August. *Distributed cognition as a theoretical framework for HCI: Don't throw the baby out with the bathwater – the importance of the cursor in air traffic control.* Report 9403, Department of Cognitive Science, University of California, San Diego.

Hollan, James, Edwin Hutchins, and David Kirsh. 2000. "Distributed Cognition: Toward a New Foundation for Human-Computer Interaction Research." *ACM Transactions on Computer-Human Interaction*, 7(2): 174–196.

Hutchins, E. & Palen, L. 1997. "Constructing Meaning from Space, Gesture and Speech." In: L.B. Resnick, R. Saljo, C. Pontecorvo, & B. Burge, (eds.) *Discourse, Tools, and Reasoning: Essays on Situated Cognition*. Springer-Verlag, Heidelberg, Germany. Pp. 23-40.
Hutchins, E. 1994. *Cognition in the Wild*. MIT Press, Cambridge, MA.
Hutchins, E. 1995. "How the cockpit remembers its speed." *Cognitive Science* 19:265-288.

Newell, Allen and Simon, Herbert, 1976. "Computer Science as Empirical Inquiry: Symbols and Search". *Communications of the ACM*, 19(3).

Norman, D.A. 1998. *The Design of Everyday Things*. The MIT Press.

Tomasello, M., Carpenter, M., Call, J., Behne, T. and Moll, H. Understanding and sharing intentions: The origins of cultural cognition. 2005. *Behavioral And Brain Sciences* 28: 675–735

7.8. Other cited sources

Derrida, Jacques. 1997. *Of Grammatology*. Baltimore: John Hopkins University Press.

Fehér, Márta, Zemplén, Gábor, Binzberger, Viktor (eds). 2006. *Értelem és történelem (Understanding and History), Tudománytörténeti és Tudományfilozófiai évkönyv*, II. kötet. Budapest: L'Harmattan.

Hofstadter, D. R. 1979. *Godel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books

Mander, Mary S. 1987. "Bordieu, the Sociology of Culture and Cultural Studies: A Critique." European Journal of Communication, 2: 427-53.

Margitay, Tihamér. 2006. "Tudományelmélet és tudománytörténet viszonya ismeretelméleti szempontból." Pp. 155-178 In: Fehér, Márta, Zemplén, Gábor, Binzberger, Viktor (eds). 2006. *Értelem és történelem (Understanding and History), Tudománytörténeti és Tudományfilozófiai évkönyv*, II. kötet. Budapest: L'Harmattan.

Pirsig, R. 1984 Zen and the Art of Motorcycle Maintenance. Bantam Books.

Maturana, H. and Varela, F. 1980. *Autopoiesis and Cognition. A Realization of the Living*. Dordrecht: Reidel Publishing Company.