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

Viktor Binzberger
Béla Julesz Cognitive Science Research Group
Budapest University of Technology and Economics and The Hungarian Academy of Sciences

Abstract
This paper 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 language use, this general interpretation permeates the particular practices of understanding that are possible within the language framework.

I support my argument by contrasting the hermeneutic concerns about the understanding between programmers which stand behind the design of the Ada and the Python programming languages. Ada, with its emphasis on achieving seamless communication through rationalistic standardization and the 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 a culture of open-ended discussion and self-reflective practices of conventionalization that is characteristic of the FLOSS world.

Keywords: hermeneutics, programming, practices

Hermeneutics, software technology and understanding

During the process of software development, participants – users, developers, customers – must engage in various *hermeneutic practices* to achieve a shared understanding with respect to many issues: software artifacts, program codes, relative valuations of problems, norms, and the general social context of development. These practices can take a number of forms, including producing inscriptions in formal or informal languages, or engaging in multi-modal discussions in various media. 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 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 the explicit symbolic inscriptions – like source codes – can be interpreted. What "gets transmitted" through them is often not an explicit message, but unarticulated 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 lifeworld, 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 everyday practices of understanding that are possible within the language framework.*

I'm going to 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, 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 some influential philosophers of the hermeneutic tradition. Besides providing an interesting insight into the practices of Cold War-era DoD-funded military projects, this choice brings an opportunity to assess the claims of Heidegger and others 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 a widely used open-source language, Python, because it seems to follow a different strategy to address similar concerns. 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 that is characteristic of the FLOSS world. It will be shown that the classical analyses of hermeneutic philosophers are not adequate in this cultural horizon. To understand the enthusiasm of Python developers, we have to turn toward philosophers who raise similar concerns, but argue for a positive appropriation of technology. Python programmers are influenced by Robert Pirsig and his 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 cooperative work.

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 us review the classical standpoints in the hermeneutic tradition about the status of technical language.

The problem of the "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 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 the argument based on the wider context of Heidegger's work, it can be summarized as the following: Since language permeates our practices and our understanding of the world and ourselves, and also given that modern technology employs a range of controlled languages, 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 or solve problems through programming, 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 (presumably[1]) 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 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. How can this be reconciled with his hermeneutist stance? 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 impact of info-communication technologies on our lifeworld is undeniable, these philosophers are overshadowed by their generalized pessimistic and deterministic overtones, which are not widely shared by philosophers of technology anymore. Up to the present 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 free flow of information. Yet still, in their outmoded, essentialist fashion, these classics all address a very profound question, one that is still relevant today: *how do people understand each other and themselves 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 *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 also 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 IT. 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 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!

Concerns behind the design of the Ada programming language

The design process of Ada was situated in a discourse in quest for the "best" language, in terms of programmer productivity, reliability and 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 languages among the various development fields. The crisis was often attributed to the lack of expressivity of the languages used, and the difficulty of reusing proven solutions. 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 by consolidating all DoD development under a unified language. However, the committee went much further: 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. [T]he programmer [should] 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. (WOODENMAN - Needed Characteristics, 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. (WOODENMAN - Needed Characteristics, reproduced by Whitaker, 1993)

With these criteria, the language designers tried to address certain situations, where understanding among programmers often breaks down. The programming language serves not only as an interface with the computer, but also as the linguistic medium of the programmer

community, in which they articulate their problems, intentions, assumptions and ideas. The occasional misunderstandings often result in reliability failures. For example, maintaining and fixing a program code that was written by someone else requires a lot 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 necessary because of the lack of "commonalities", or common conceptual abstractions covering a wider set of situations. In order to make code "reusable" in future problem situations, the common structures and conventions need to be standardized and the code has to be divided into independent functional modules. Specific language structures were proposed to alleviate these issues, to allow for a "higher abstraction level" (Smith, 1987).

What is common in all these breakdowns of understanding is that they lead to open-ended hermeneutic efforts. These hermeneutic episodes are pictured by the language designers as unwanted, because they introduce unpredictable outcomes and delays, thereby constitute risk.

The belief that there exists a totally transparent and explicit formulation of any problem reveals a certain epistemological naiveté from the part of the language committee. Such beliefs have been contested by many theorists, because when we start to explicate our background knowledge, we implicitly start to build on an even broader set of background knowledge, again in need of explication (Winograd and Flores, 1987).

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 among the programmer's priorities if she doesn't have any feedback on the long-term costs of her code (Kling and Scacchi, 1979: 37). Hard-to-decipher code can easily make her the irreplaceable "key figure" of the project (Boehm, 1979), by the fact that only she can understand it. 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). How could she be motivated to think abstractly and modularly, to write code not for herself, but for her successors, who have to interpret it (Smith, 1987)? Or framing the question within the socio-cultural space: how can her thinking be aligned with the abstract principles and long-term interests of the management (Gerhardt, 1989; Kling and Scacchi, 1979)?

In order to answer this question, we have to see that the Ada initiative is an attempt to thoroughly *transform the way in which programming problems are perceived and articulated by the programmer*. The concerns of the management are carefully designed right into the structure of the language. From now on, the programmer can't even conceptualize her problem without taking these into account, 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. This transformation took place at the level of language skills and in the patterns of interaction.

What makes the programmer follow the 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, technically 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.

It works as an abstract "electric fence", forcing the programmer to think the right way. *The compiler is thus a political artifact, working as an obligatory passage point* (Lessig, 1999; Winner, 1986; 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 compiler implementations that do not fulfill the Ada specifications.

Even within an established language framework, there is room for misunderstanding as the various participants come with different backgrounds of understanding. To address this, the structure of Ada reflects the organizational hierarchy of development at an even more specific level. The organization of functional modules (packages) is supposed to mirror the hierarchical organization of work by delineating self-enclosed units that can be intellectually managed by a responsible individual or a team (Ichbiah, 1984: 994). The pathways of communication and control between the modules – and thus between associated developers – have to be declared explicitly in the code. The goal is to achieve a certain economy of interaction between developers. The individuality of backgrounds doesn't matter so much if the space of possible/necessary interactions between developers is reduced and formalized to a certain extent. This induces a certain social stratification, a power-hierarchy between designer and developer groups. It is ironic that the proverb "divide and conquer", used often by design theorists to refer to modularization (e.g. DeRemer and Kron, 1975), is at the same time a management strategy in the original sense!

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. However, for the programmer, this built-in perspective rarely ever gets into the focus of thematic understanding. She keeps her immediate 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. In fact, 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 her technological lifeworld.

The designers of the programming language thus implement a modernistic tendency: they draw a line 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. 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)

In the utopian world of the Ada designers, programmers won't need to make individual decisions between alternative interpretations or practices of language use, because others will have already taken care of these. The user is not allowed to make changes to the linguistic framework – she has no choice but to obey the syntactical and stylistic rules of the language. This *disburdening* is in essence a military hierarchy embodied in the language. In this ideal world, no special ability is needed for the frictionless development, and the place of the individual programmer can be filled by *anyone*. The once-admired hacker gives place to the 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* were necessary to build such highly complex technological systems as the F-16 jet fighter (Whitaker, 1993) – and they are still relevant in most contemporary software development organizations.

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 technical code, which reduces language to the purpose of the coordination of human action. But this is not the conclusion I'd like to draw. First, it is not a reified „language machine" or "Technology" with a capital "T" what "masters the essence of the human being", but it is rather a collective act of managers, programmers and political decision-makers, who are 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 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.[2]

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

Nowadays Ada contributes much less than 1% to the software developed worldwide. There is also a wide proliferation of 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. The "sofware crisis" persisted despite the advent of Ada, not only in the DoD but also in the private sector. More detailed criticism can be found in (Baker, 1997; Bennett et al., 1982; Dijkstra EDW658-663; Feinberg, 1987), which argue that Ada is too bureaucratic and inefficient.

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 language cannot be extended effectively killed all 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 language designers.

So far, the theories of Heidegger, Habermas and Márkus seem to be very consistent with the design rationales built into Ada. Their fear is exactly what Ada aimed 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.

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 always 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.

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. 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), and they were introduced in order to be cited (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 – 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, 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 explicate 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 the "Zen of Python" 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 data typing system is not static – as in Ada – but dynamic, it is not easy to tell, for example, what kind of parameters can be used to call a function. 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 through other channels. This means that in principle, it is quite easy to write incomprehensible Python programs. It can be made very hard to guess from the source code what goes on at runtime.

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. Reliability concerns are addressed through promoting peer review and extensive testing, rather than language constructs.

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, you write understandable programs because you are disciplined by the compiler (and the legal code behind it). In the Python community, "readability" is influenced by syntax, but 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
>
> 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! (PEP 8: "Style Guide for Python Code")

This approach 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 sometimes results in swift changes of the core language (Rossum, 2001).

On 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 cast using the Apache Project voting scheme (See figure 1).

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). 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 a single debate. It is also worth pointing out that the frequencies of the postings using these words are correlated.

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"*.

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.

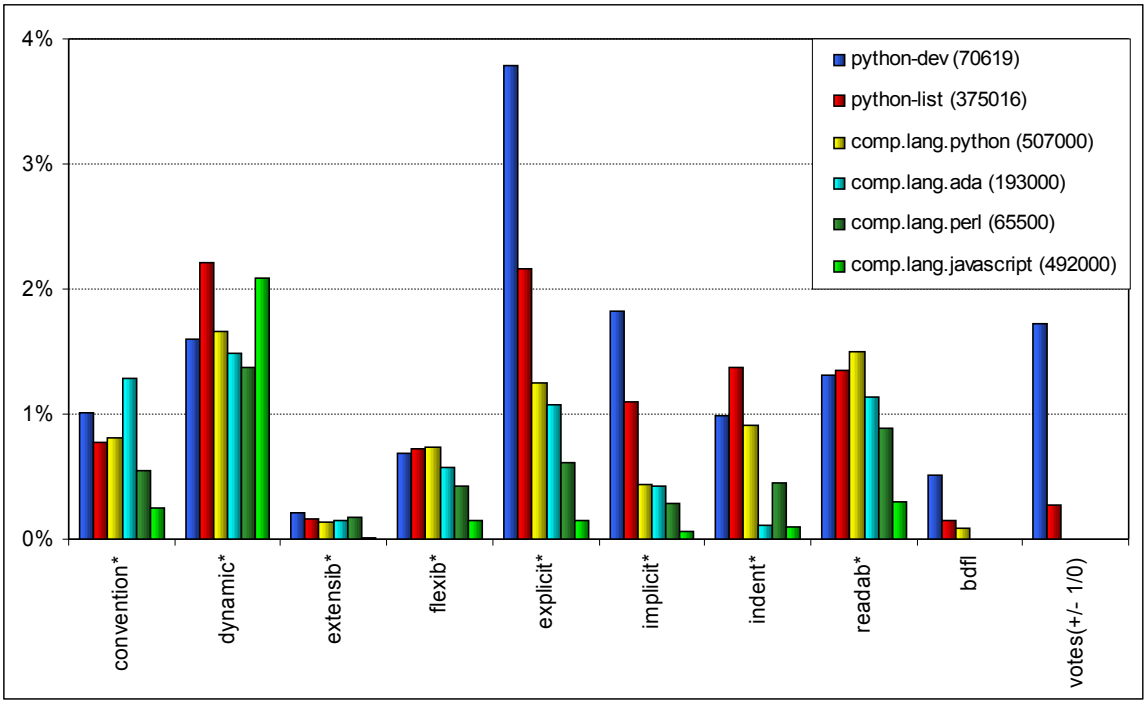

Fig. 1. Percentage of mailing list/newsgroup postings containing the respective keywords. (Figures for python-dev and python-list refer to occurrences in non-cited body text, whereas comp.lang. newsgroup figures contain all occurrences. The total number of postings is shown in parentheses.)*

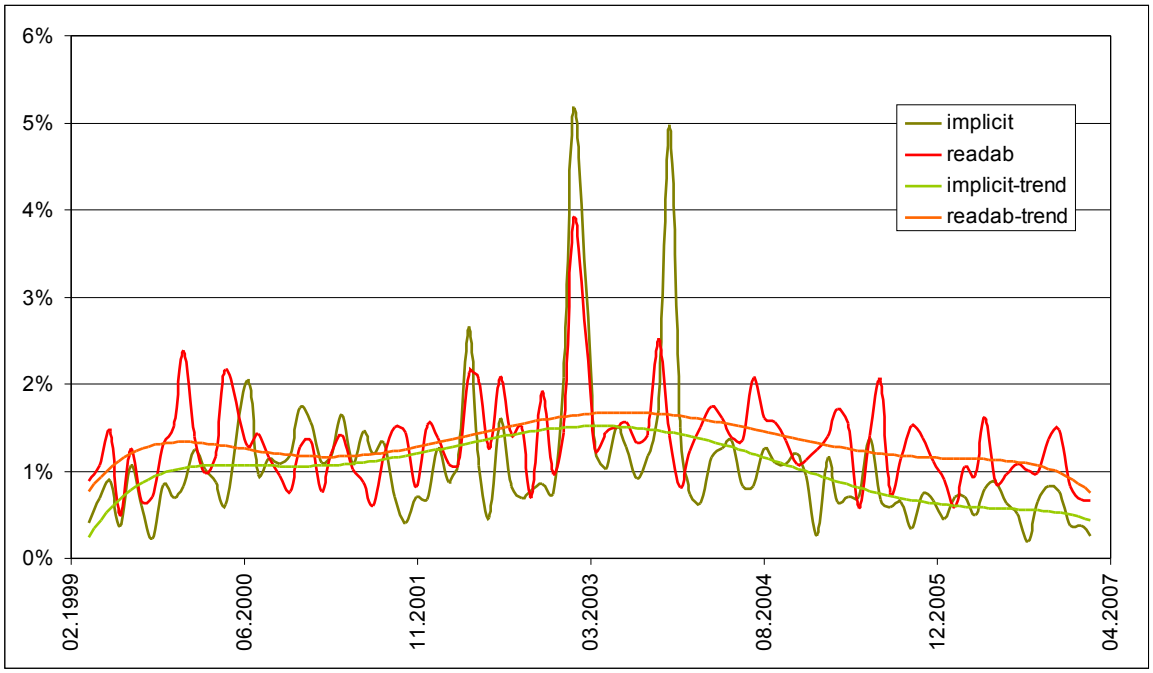

Fig. 2. Correlation between the relative frequency of pattern "implicit" and "readab" in python-list postings. The correlation coefficient is 0.502.*

Conclusions drawn from the Python case

We can now turn toward 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 well-designed technical artifacts do (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. 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 is not necessarily "anarchic". Instead of having strict methodologies and controlled technical codes, their conventions emerge in rather highly structured, decentralized and self-disciplined hermeneutic practices. 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)

If DeMarco and Lister are right, the "rigidity" inherent in the concept of the "technical code" is self-defeating: socio-technical systems conditioned by overly rigid technical codes might not have the necessary "self-healing" quality to survive in a fast-changing world. In contrast, the Python language and the community itself, by having a "sufficient degree of natural ad-hocracy", can accommodate itself fast to new challenges. Certain conventions 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.

General conclusions

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 elegance, but this would have exceeded the limits of this article.

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 some of its core features from Ada. Nevertheless, the biases built into the instruments can be traced back to the reflections of users and designers, and can be situated in the cultural horizon.

This is consistent with my claim 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 assumptions (as he thinks), it is the paradigmatic and specialized nature of *ongoing hermeneutic practice* that grants the shared background.

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 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 many alternatives. What I still find relevant is *the general framing of their questions*: the emphasis on the praxis-constituting role of language and on the importance of hermeneutic practices in our technological culture.

Acknowledgments
I'd like to thank the BME-HAS Béla Julesz Cognitive Science Research Group and the BME Department of History and Philosophy of Science for their support in my research.

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;
         -- issue prompt until non-blank responses

         Text_IO.New_Line;
      end loop Get_Response;
   end Yes;
   -----------------------------------------------------------------
   […]
end Terminal_IO;
```

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()
        for c in response:
            if c not in ' \n':
                return (c == 'y') or (c =='Y')
```

Sources relevant for the Ada case

Comprehensive archive of Ada-related documents
http://www.iste.uni-stuttgart.de/ps/AdaBasis/pal_1195/ada/ajpo/

Websites dedicated to Ada-related activity
http://www.acm.org/sigada/
http://www.adahome.com/
http://www.adaic.com/
http://www.ada-europe.org/

FAQ about the closure of the Ada Joint Program Office
http://sw-eng.falls-church.va.us/ajpofaq.html

Edsger W. Dijkstra Archive
http://www.cs.utexas.edu/users/EWD/transcriptions/transcriptions.html

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."Online: http://archive.adaic.com/docs/flyers/83cplus.html.

Carlson, W. E., L.E. Druffel, D.A. Fisher, and W.A. Whitaker. 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, R. 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., and H. Kron. 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., G.P.B. Barnes, R.J. Firth, and M. Woodger. 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.

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. Online: 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.

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., B.D. Kornman, and J.R. Wilson. 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). Online: 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 W. Scacchi. 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.

Sources relevant for the Python case

Policies, Processes and Guidelines

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

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

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

Criticism and Assessment

Arnold, Ken. 2004. "Style is Substance." Online: http://www.artima.com/weblogs/viewpost.jsp?thread=74230 (Accessed: jan. 31, 2007).
Bicking, I. 1995. "UnZen of UnPython." Online: http://blog.ianbicking.org/unzen-of-unpython.html (Accessed dec. 1, 2006).
Rossum, G.v. 1996. "Foreword." In Programming Python, 1st ed., Mark Lutz (ed.). O'Reilly & Associates Online: http://www.python.org/doc/essays/foreword.html
Rossum, G. v. 2001. "Foreword." In Programming Python, 2nd ed., Mark Lutz (ed.). O'Reilly & Associates. Online: http://www.python.org/doc/essays/foreword2.html
Rossum, G. v. 2006. "Language design is not just solving puzzles." Online: http://www.artima.com/weblogs/viewpost.jsp?thread=147358.

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

Sources on software methodology

Beck, K. 1999. *Extreme programming explained: Embrace change.* 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., and R. Turner. 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., and T. Lister. 1999. *Peopleware,* 2nd ed. New York: Dorset House Publishing.

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

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

Other sources

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 Computationalism: New Directions, Matthias Scheutz (ed.). Cambridge: MIT Press.

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

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

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

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

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

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

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.

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

Feenberg, A. 1991. *Critical Theory of Technology*. New York: Oxford University Press.

Feenberg, A. 1995. "Subversive Rationalization: Technology, Power, and Democracy." In *Technology and the Politics of Knowledge*, Feenberg & Hannay (eds.). 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." In *Technology and the Good Life*, Higgs, Strong, Light (eds.). University of

Chicago Press.
Feenberg, A. 2003. "Critical Evaluation of Heidegger and Borgmann." In *Philosophy of Technology: An Anthology*, Scharff & Dusek (eds.). Blackwell Publishing.
Foucault, M. 1982. "The subject and power." In *Michel Foucault: Beyond Structuralism and Hermeneutics*, H. Dreyfus and P. Rabinow (eds.) . 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 *Habermas On Society and Politics: A Reader*, Seidman (ed.). Boston: Beacon Press.
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. Online: http://www.focusing.org/apm_papers/heelan.html (Accessed dec. 1, 2006)
Heidegger, M. 1950. "Das Ding." In *GA 7*. Vittorio Klostermann.
Heidegger, M. 1957. *Hebel - Der Hausfreund*. Pfullingen: Günther Neske.
Heidegger, M. 1962. Die Technik und die Kehre. Pfullingen: Günther Neske.
Heidegger, M. 1996. [BT] *Being and Time: A Translation of Sein and Zeit*, J. Stambaugh (trans.). New York: State University of New York 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 *GA 12*. 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, D. 1990. "Program One. A Phenomenology of Technics." In *Technology and the Lifeworld: From Garden to Eden.* Bloomington: Indiana University Press.
Ihde, D. 1999. *Expanding Hermeneutics: Visualism in Science.* Northwestern University Press.
Ihde, D. 2003. "Heidegger's Philosophy of Technology." In *Philosophy of Technology: An Anthology*, Scharff, Dusek (eds.). Blackwell Publishing.
Hofstadter, D. R. 1979. *Godel, Escher, Bach: An Eternal Golden Braid*. New York: Basic Books
Kochan, J. 2005. *A Poetics of Tool-use - Explorations in Heidegger and Science Studies*. PhD thesis, Churchill College.
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 *Shaping Technology / Building Society: Studies in Sociotechnical Change*, Bijker and Law (eds.). 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.
Lessig, L. 1999. *Code and Other Laws of Cyberspace.* New York: Basic Books
Licklider, J.C.R. 1960 (March). *IRE Transactions on Human Factors in Electronics,* volume HFE-1: 4-11.
Márkus, G. 1987. "Why is There No Hermeneutics of Natural Science? Some Preliminary Theses." *Science in Context* 1: 5-51.
Norman, D.A. 1998. *The Design of Everyday Things*. The MIT Press.
Pirsig, R. 1984. *Zen and the Art of Motorcycle Maintenance.* Bantam Books.
Polányi, M. 1962. "The Republic of Science: Its Political and Economic Theory." *Minerva* 1: 54-74.
Rouse, J. 1999. "Understanding Scientific Practices: Cultural Studies of Science as a Philosophical Program." In *The Science Studies Reader*, Biagioli (ed.). New York,

London: Routledge.
Schwendtner, T., L. Ropolyi, and O. Kiss (eds.). 2001. *Hermeneutika és a természettudományok* (Hermeneutics and the Natural Sciences). Budapest: Áron kiadó.
Shapiro, G., and A. Sica. 1984. *Hermeneutics: Questions and Prospects.* Amherst.
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." In Representation in Scientific Practice, M. Lynch and S. Woolgar (eds.). Kluwer Academic Publishers.
Torres-Gregory, W. 1998. "Heidegger On Traditional Language And Technological Language." Presentation on the 20th World Congress on Philosophy.
Turkle, S. 1999. "What Are We Thinking about When We Are Thinking about Computers?" In *The Science Studies Reader,* Biagioli (ed.). New York, London: Routledge.
Winner, L. 1986. "Do Artifacts Have Politics?" In *The Whale and the Reactor*. Chicago: Univ. of Chicago.
Winograd, T. and F. Flores. 1987. *Understanding Computers and Cognition*. Norwood, NJ: Ablex Corporation.

Endnotes

1 For a detailed critical reconstruction, see Feenberg 1996.
2 http://www.adacore.com/2006/05/01/hamilton-sundstrand-selects-gnat-pro-for-boeing-787-air-conditioning-pack-control-unit/