TECHNOLOGY MADE LEGIBLE

A Cultural Study of Software as a Form of Writing in the Theories and Practices of Software Engineering

Federica Frabetti
Goldsmiths College, University of London

Submitted for the degree of PhD

2009

Abstract

My dissertation proposes an analytical framework for the cultural understanding of the group of technologies commonly referred to as 'new' or 'digital'. I aim at dispelling what the philosopher Bernard Stiegler calls the 'deep opacity' that still surrounds new technologies, and that constitutes one of the main obstacles in their conceptualization today. I argue that such a critical intervention is essential if we are to take new technologies seriously, and if we are to engage with them on both the cultural and the political level.

I understand new technologies as technologies based on software. I therefore suggest that a complex understanding of technologies, and of their role in contemporary culture and society, requires, as a preliminary step, an investigation of how software works. This involves going beyond studying the intertwined processes of its production, reception and consumption – processes that typically constitute the focus of media and cultural studies. Instead, I propose a way of accessing the ever present but allegedly invisible codes and languages that constitute software. I thus reformulate the problem of understanding software-based technologies as a problem of making software legible.

I build my analysis on the concept of software advanced by Software Engineering, a technical discipline born in the late 1960s that defines software development as an advanced writing technique and software as a text. This conception of software enables me to analyse it through a number of reading strategies. I draw on the philosophical framework of deconstruction as formulated by Jacques Derrida in order to identify the conceptual structures underlying software and hence 'demystify' the opacity of new technologies.

Ultimately, I argue that a deconstructive reading of software enables us to recognize the constitutive, if unacknowledged, role of technology in the formation of both the human and academic knowledge. This reading leads to a self-reflexive interrogation of the media and cultural studies' approach to technology and enhances our capacity to engage with new technologies without separating our cultural understanding from our political practices.

Chapters Outline

1. From Technical Tools to Originary Technicity: The Concept of Technology in Western Philosophy

In this chapter I suggest that the problem of 'new technologies', and of the kind of knowledge that can be produced about them, cannot be addressed without radically reconsidering what we mean by 'knowledge' in relation to 'technology' in a broader sense. What kind of knowledge can we produce about technology in the context of media and cultural studies, apart from the analysis of the discourses and practices of producers and consumers? What kind of knowledge are we interested in? To answer these questions, I argue that, as a preliminary step, the received concepts of technology need to be put into question. By received concepts of technology I mean the ways in which technology has been understood primarily by the Western philosophical tradition. I also argue that media and cultural studies can highly benefit from a productive dialogue with philosophy on the subject of technology.

I outline two traditions of philosophical thought about technology: the dominant Aristotelian conception of technology as an instrument, and an alternative line of thought based on the concept of the 'originary technicity' of the human being. Subsequently, I draw on the work of thinkers belonging to the latter tradition (Martin Heidegger, Bernard Stiegler, André Leroi-Gourhan, Jacques Derrida) to develop a different way of thinking about technology, one that will ultimately prove more productive for my investigation of software.

2. Language, Writing and Code: Towards a Deconstructive Reading of Software

This chapter deals with the concept of 'writing' in relation to 'language' and 'code', and with the usefulness of these concepts for the understanding of software. It asks to what extent and in what way software is legible, and it proposes a deconstructive reading strategy of software.

I start this chapter with a discussion of the use of 'language', 'code' and 'writing' as explanatory concepts for software-based technologies. I draw mainly on N. Katherine Hayles' understanding of the 'regime of code' as opposed to the regimes 'of speech' and 'of writing', and on her suggestion that writing and code are

intertwined in software (Hayles 1999, 2002, 2005). Nevertheless I question her assumption that software, as a technical object, is 'beyond metaphysics'.

I subsequently take recourse to Derrida's understanding of writing as a material practice to point out that every code is material, and propose that, to understand software, we need to engage in a deconstructive reading of software.

3. Software as Material Inscription: The Beginnings of Software Engineering

In Chapter Three I show how, at the end of the 1960s, Software Engineering established itself as a discipline through the attempt to control the constitutive fallibility of software-based technology. I develop a close reading of the technical literature related to the first two conferences on Software Engineering ever held, the NATO Conference on Software Engineering held in Garmisch (Germany) in 1968 and the one held in Rome in 1969. These conferences, which were attended by many of Software Engineering's 'founding fathers', are considered the foundational moment of the discipline. The conferences are very well documented. What is more interesting is that all the most controversial issues concerning software's ontological status and its ways of working were already discussed at that event. In this chapter I argue that in the foundational texts of Software Engineering 'software' was constituted as a process of material inscription through the continuous undoing and redoing of the boundaries between 'software' itself, 'writing' and 'code'.

4. From the Cathedral to the Bazaar: Software as the Unexpected

In this chapter I investigate how the mutual co-constitution of 'software', 'writing' and 'code' develops in the Software Engineering of the 1970s and 1980s. I argue that Software Engineering gradually emerges as the locus for a continuous suspension and reconfirmation of the instrumentality of software. I also show how software escapes its own definition as a 'tool' by generating unforeseen consequences, one of which is the emergence of open source programming in the 1990s. I then examine two of the fundamental texts on the management of time in software development, both written by Frederick Brooks in the mid-1970s and mid-1980s respectively. Subsequently, I investigate a classic of Software Engineering in the 'open source era', Eric Steven Raymond's *The Cathedral and the Bazaar*, which responds to Brooks' theories from the point of view of the mid-1990s.

5. Writing the Printed Circuit: For a Genealogy of Code

In this chapter I investigate the process of linearization at work in programming languages, in order to question current understandings of digitization and digitality.

I start by focusing on the position of programming languages and formal notations within Software Engineering. I then examine two classical works by J. D. Ullman on programming languages and compilers (Hopcroft and Ullman 1969; Aho and Ullman 1979) and discuss the concepts of formal notation, formal grammar, programming language and compiler. I concentrate on the process of 'compiling' - that is, the process through which a program is made executable through its 'translation' from a high-level programming language into machine-language, and ultimately into physical transformations that are both sequences of on/off circuits and streams of 0s and 1s. I also investigate the role of the 'blank' in the text of the program and the way in which the blank originates discreteness in time (that is, at the time of the execution of the program).

Drawing on Derrida's understanding of the materiality of signification, I argue that the distinction between 0 and 1 is always already material, since it is enabled by the mark. By considering the distinction between 0 and 1 as 'material transcendence' I break free of the dilemma regarding the ontological status of code (Hayles 2005). I also propose we understand the distinction between 0 and 1 as externalized memory, and I investigate the implications of thinking this distinction as the constitution of time *in/as* software.

Contents

Acknowledgements

This thesis has been inspired by almost fifteen years of 'writing' software as a Software Engineer for telecommunications. The people I wrote software with are too many to be named; nevertheless, I would like to thank them here because I learnt so much from them while also having so much fun.

Goldsmiths, University of London, has provided an exciting and stimulating environment for the much more critical questioning of technology over the past few years. First and foremost I am grateful to Joanna Zylinska for being an exceptional supervisor, a demanding interlocutor, an inspiring guide and a friend. More often than not she seemed to know where I was heading before I did, which made me work really hard in order to surprise her. Her work remains for me a model of intellectual and political engagement. I have also been enriched by feedback and comments from many other people in the Department of Media and Communications, in particular from Sarah Kember. Janet Harbord and David Morley have also provided insightful and provocative readings of my work.

In 2003 I had the privilege to discuss the early stages of my project with N. Katherine Hayles at a Masterclass organized by Rosi Braidotti at the University of Utrecht. Although over time my thought on technology has taken a different direction from Hayles', I maintain the greatest admiration for her work and feel fortunate to have been in dialogue with her.

I also want to express my gratitude to Gary Hall for his generous comments on my work on so many occasions and for sharing with me some of his tremendous insights on technology (not least the one about 'quasi-malfunctioning' software).

Very special thanks are due to Liana Borghi at the University of Florence for being my first mentor and an inestimable friend. She has changed my life in more than one way and continues to do so.

Many friends in Italy and in England have stayed assiduously in touch and have put up with my constant drifting around Europe. I am particularly grateful to Amy Elizabeth Barksdale (for her unfailing emotional support and extraordinary sense of humour), Dilek Basgurboga (for being a friend and a sister throughout), Sandra Gaudenzi (for memorable discussion and a couple of very enjoyable trips to Northern Europe) and Laura Cellini (whose life-long friendship I treasure). Special thanks go to Roberto Camano for providing priceless feedback on grammars and compilers - and for much more.

Through thick and thin over these years I have had the close support of a superb team: my gratefulness goes to Rosie Harman, David Mabb, Black One and Other One for seeing me through all this with so much love, wit and understanding.

A big thank you to Emma Inch for sharing with me monsters and aliens - and for bringing so much love and beauty into my life.

Finally, I want to express all my gratitude to my feisty and fiercely humorous mother Carla for her unconditional support. She and my late grandmother have been models of determination, optimism and cheerfulness throughout my life.

This thesis is dedicated to my mother with love.

Introduction

This thesis is inspired by my experience of more than a decade of designing software for telecommunications for a number of companies across two continents. I was fortunate enough to have witnessed, and made a small contribution to, the birth of the second generation of mobile telephony, or GSM (a geological stratum of current G3/UMTS).[1] In the early 1990s I wrote the SS7 TCAP (Transaction Capabilities Application Part) protocol for Italtel-Siemens telephone exchanges and enjoyed a protracted struggle with C language, UNIX, a few types of Assembler languages and a range of European and world-wide standards and recommendations. I also experienced the expansion of digital mobile telephony into Russia and China in the early to mid-1990s and developed software for SMS (Short Message Service) at a time when nobody used the term 'texting' and when adding text messaging to mobile communications was considered by many (including myself) an unpromising idea.[2]

---

[1] GSM was originally a European project. In 1982, the European Conference of Postal and Telecommunication Administration (CEPT) instituted the Groupe Spécial Mobile (GSM) to develop a standard for a mobile telephone system that could be used across Europe. In 1989 the responsibility for GSM was transferred to the European Telecommunications Standards Institute (ETSI). GSM was re-signified as the English acronym for Global System for Mobile Communications and Phase One of the GSM specifications were published in 1990. The world first public GSM call was made on 1 July 1991 in a city park in Helsinki, Finland, an event which is now considered the birthday of second-generation mobile telephony – the first generation of mobile telephony to be completely digital. In the early 1990s Phase Two of GSM was designed and launched, and GSM rapidly became the world-wide standard for digital mobile telephony. A decade later it led to the third-generation mobile telecommunication systems – the Universal Mobile Telecommunication System (UMTS) (Kaaranen et al. 2005).

[2] TCAP is a digital signaling system that enables communication between different parts of digital networks, such as telephone switching centers and databases. SMS is a communication service standardized as part of the GSM network (the first definition of SMS is to be found in the GSM standards as early as 1985), which allows for the exchange of short text messages between mobile telephones. The SMS service was developed and commercialized in the early 1990s. Today SMS text messaging is the most widely used data application in the world.

However, over time I started questioning my own engagement with technology. Perhaps a mix of my background in the humanities, my general wariness of the corporate environment and the political commitment to 'think differently' that came from my involvement with the Italian left, with the unions and also with the queer movement throughout the 1990s made me quite conscious of the limits of the merely technical understanding of technology. Thus, this thesis also stems from my desire to ask different questions of technology from those posed by the technical environment in which I had an opportunity to work. In 2004, when I first began investigating the nature of software from a non-technical point of view, the context of media and cultural studies presented the most interesting academic framework for such an enquiry - although I have actually ended up questioning media and cultural studies' approach to technology.

The principal aim of this thesis is to propose an analytical framework for the cultural understanding of the group of technologies commonly referred to as 'new' or 'digital'. Taking into account the complexity and shifting meanings of the term 'new technologies', I understand these technologies as sharing one common characteristic: they are all based on software. The currency of the term 'software' in public and academic discourses does not yet match that of 'new technologies', although its appearance is more and more frequent. However, its meaning seems to be equally shifting and unclear. I argue that, in order to understand what new technologies are, we need first of all to focus on what software is.

Therefore, throughout the course of my argument I primarily ask the question 'what is software?'. I argue that this question needs to be dealt with seriously if we want to begin to appreciate the role of technology in contemporary culture and society. In other words, I call for a radical 'demystification' of new technologies through a demystification of software. I also maintain that in order to understand new technologies we need first of all to address the mystery that surrounds their functioning and that affects our comprehension of their relationship to the cultural and social realm. This will ultimately involve a radical rethinking of what we mean by 'technology', 'culture', and 'society'.

The reason I believe such an intimate engagement with new technologies is needed is not only related to their pervasiveness in our world, but also to the fact that we are constantly being asked or even forced to make decisions about software-based technologies in our everyday life: for instance, when having to decide whether to use commercial or free software, when to upgrade our computer, or whether we should own one at all. I am particularly interested in the political significance that our – conscious and unconscious - involvement with technology carries.

Therefore, with this thesis I also seek a way to think new technologies politically. More precisely, I argue that the main political problem with new technologies is that they exhibit - in the words of Bernard Stiegler – a 'deep opacity' (Stiegler 1998: 21). As Stiegler maintains, 'we do not immediately understand what is being played out in technics, nor what is being profoundly transformed therein, even though we unceasingly have to make decisions regarding technics, the consequences of which are felt to escape us more and more' (21).[3] I suggest that, in order to develop political thinking about new technologies, we need to start by tackling their opacity.[4]

To be able to elaborate further on what I mean by demystifying the opacity of new technologies, and particularly of software, let me start by examining the place of new technologies in today's academic debate.

---

[3] We can provisionally assume here that the word 'technics' (belonging to Stiegler's partially Heideggerian philosophical vocabulary) indicates in this context contemporary technology, and therefore includes what I am referring to here as 'new technologies'.

[4] Alfred Gell (1992) develops an interesting reflection on the relations between art, technology and magic. Drawing on the example of Trobriand canoe-boards, Gell argues that the foundation of art is a technically achieved level of excellence that a society misrepresents to itself as a product of magic. Gell views art as a special form of technology and art objects as devices to obtain social consensus. For Gell 'the power of art objects stems from the technical processes they objectively embody: the *technology of enchantment* is founded on the *enchantment of technology*' (Gell 1992: 44). The magical prowess, which is supposed to have entered the making of the art object, depends on the level of cultural understanding that surrounds it. The same can be said of technology: 'the enchantment of technology is the power that technical processes have of casting a spell over us so that we see the real world in an enchanted form' (44). This is what Gell names the 'halo effect of technical difficulty' (Gell 1992: 48; Pinney and Thomas 2001: 3). He argues that art objects are made valuable precisely by virtue of the 'intellectual resistance' they offer to the viewer (49) and that 'technical virtuosity is intrinsic to the efficacy of works of art in their social context' because it creates an asymmetry in the relationship between the artist and the spectator (52). Gell's reflection contributes to show the cultural character of the sense of enchantment that surrounds technology. Furthermore, by suggesting that such sense of enchantment has a role in the creation of social consensus, it provides evidence to the fact that any attempt to change the processes through which we understand technology has significant political consequences.

New technologies are an important focus of academic reflection, particularly in media and cultural studies. With the formulation 'media and cultural studies' I mean to highlight that the reflection on new technologies is positioned at the intersection of the academic traditions of cultural studies and media studies. Nevertheless, to think that technology has only recently emerged as a significant issue in media and cultural studies would be a mistake. In fact, I argue that technology (in its broadest sense) has been present in media and cultural studies from the start, as a constitutional concept. The intertwining between the concepts of 'medium' and 'technology' dates back to what some define as the 'foundational' debate between Raymond Williams and Marshall McLuhan (Lister 2003: 74). While a detailed discussion of this debate would divert us from the main focus of this introduction, it must be noticed that in his work McLuhan was predominantly concerned with the technological nature of the media, while Williams emphasized the fact that technology was always socially and culturally shaped. At the risk of a certain oversimplification, we can say that British media and cultural studies has to a large extent been informed by Williams' side of the argument – and has thus focused its attention on the cultural and social formations surrounding technology, while rejecting the ghost of 'technological determinism', and frequently dismissing any overt attention paid to technology itself as 'McLuhanite' (Lister 2003: 73). Yet technology has entered the field of media and cultural studies precisely thanks to McLuhan's insistence on its role as an agent of change.

One must be reminded at this point that, in the perspective of media and cultural studies, to study technology 'culturally' means to follow the trajectory of a particular 'technological object' (generally understood as a technological product), and to explore 'how it is represented, what social identities are associated with it, how it is produced and consumed, and what mechanisms regulate its distribution and use' (DuGay 1997: 3). Such an analysis concentrates on 'meaning', and on the way in which a technological object is made meaningful. Meaning is understood as not arising from the technological object 'itself', but from the way it is represented in the discourses surrounding it. By being brought into meaning, the technological object is constituted as a 'cultural artefact' (10). Thus, meaning emerges as intrinsic to the definition of 'culture' deployed by media and cultural studies. This is the case in Williams' classical definition of culture as a 'description of a particular way of

life', and of cultural analysis as 'the clarification of the meanings and values implicit and explicit in particular ways of life' (Williams 1961: 57), as well as in a more recent understanding of 'culture' as 'circulation of meanings' (a formulation that takes into account that diverse, often contested meanings are produced, shared and communicated within different social groups, and that they generally reflect the play of powers in society) (Hall 1997; DuGay *et al.* 1997).

When approaching new technologies, media and cultural studies has therefore predominantly focused on the intertwined processes of production, reception, and consumption, that is on the discourses and practices of new technologies' producers and users. From this perspective, even a technological object as 'mysterious' as software is addressed by asking how it has been made into a significant cultural object. For instance, in his 2003 article on software, Adrian Mackenzie demonstrates the relevance of software as a topic of study essentially by examining the new social and cultural formations that surround it (Mackenzie 2003).[5]

Although I recognize that the above perspective remains very important and politically meaningful for the cultural study of technology, I suggest that it should be supplemented by an alternative, or I would even hesitantly say more 'direct', investigation of technology – although I will raise questions for this notion of 'directness' later on. In other words, as I have suggested above, in order to understand the role that new technologies play in our lives and the world as a whole, we also need to shift the focus of analysis from the practices and discourses *concerning* them to a thorough investigation of how new technologies *work*, and, in particular, of how software *works* and of what it *does*. Let me now explain how such an investigation of software can be undertaken.

[5] An analogous claim is made by Lev Manovich in his recent book, *Software Takes Command* (2008). In this book Manovich argues that media studies has not yet investigated 'software itself', and advances a proposal for a new field of study that he names 'software studies'. Manovich actually claims to have been the first to have used the term in 1999. He goes on to propose a 'canon' for software studies that includes Marshall McLuhan, Robert Innis, Matthew Fuller, Katherine Hayles, Alexander Galloway and Friedrich Kittler among others. However, when he speaks of 'software itself', Manovich is adamant that 'software' does not mean 'code' – that is, computer programs. For him, software studies should focus on software as a cultural object - or, in Manovich's own terms, as 'another dimension in the space of culture' (Manovich 2008: 4). Software becomes 'culturally visible' only when it becomes visual – namely, 'a medium' and therefore 'the new engine of culture' (4).

By arguing for the importance of such an investigation, I do not mean that a 'direct observation' of software is possible. I am well aware that any relationship we can entertain with software is always mediated, and that software might well be 'unobservable'. In fact, I intend to take away all the implications of 'directness' that the concept of 'demystifying' or 'engaging with' software may bring with it. I am particularly aware that software has never been univocally defined by any disciplinary field (including technical ones) and that it takes different forms in different contexts.[6] In my own study I start from a rather widely accepted definition of software as the totality of all computer programs as well as all the written texts related to computer programs. This definition constitutes the conceptual foundation of Software Engineering, a technical discipline born in the late 1960s to help programmers design software cost-effectively. Software Engineering describes software development as an advanced writing technique that translates a text or a group of texts written in natural languages (namely, the requirements specifications of the software 'system') into a binary text or group of texts (the executable computer programs), through a step-by-step process of gradual refinement (Brooks 1987; Humphrey 1989; Sommerville 1995). As Professor of Software Engineering at St Andrews University Ian Sommerville explains, 'software engineers model parts of the real world in software. These models are large, abstract and complex so they must be made visible in documents such as system designs, user manuals, and so on. Producing these documents is as much part of the software engineering process as programming' (Sommerville 1995: 4)

This formulations show that 'software' does not only mean 'computer programs'. A comprehensive definition of software also includes the whole of technical literature related to computer programs, including methodological studies on how to design computer programs - that is, including Software Engineering literature itself. The essential move that such an inclusive definition allows me to make consists in transforming the problem of engaging with software into the problem of reading it. In my thesis I will therefore ask to what extent and in what way software can be described as legible. Moreover, since Software Engineering is concerned with the

[6] For instance, a computer program written in a programming language and printed on a piece of paper is software. When such a program is executed by a computer machine, it is no longer visible, although it might remain accessible through changes in the status of the machine (such as the blinking of lights, or the flowing of characters on a screen) – and it is still defined as software.

methodologies for writing software, I will also ask to what extent and in what way software can actually be seen as a form of writing. Such a reformulation will enable me to take the textual nature of software seriously. In this context, concepts such as 'reading', 'writing', 'document' and 'text' are no mere metaphors. Rather, they are Software Engineering's privileged mode of dealing with software as a technical object. It could even be argued – as I shall throughout this dissertation - that in the discipline of Software Engineering software's technicity is dealt with as a form of writing.

As a first step it is important to notice that, in order to investigate software's readability and to attempt to read it, the concept of reading itself needs to be problematized. In fact, if we accept that software presents itself as a distinctive form of writing, we need to be aware that it consequently invites a distinctive form of reading. But to read software as conforming to the strategies it enforces upon its reader would mean to read it like a computer professional would, that is in order to make it function *as software*. I argue that reading software on its own terms is not equal to reading it functionally. For this reason, I develop a strategy for reading software by drawing on Jacques Derrida's concept of 'deconstruction'. However controversial and uncertain a definition of 'deconstruction' might be, I am essentially taking it up here as a way for stepping outside of a conceptual system while simultaneously continuing to use its concepts and demonstrating their limitations (Derrida 1980). 'Deconstruction' in this sense aims at 'undoing, decomposing, desedimenting' a conceptual system, not in order to destroy it but in order to understand how it has been constituted (Derrida 1985).[7] According to Derrida, in every conceptual system we can detect a concept that is actually unthinkable within the conceptual structure of the system itself – therefore, it has to be excluded by the system, or, rather, it must remain unthought to allow the system to exist. A deconstructive reading of software therefore asks: what is it that has to

[7] In 'Structure, Sign, and Play in the Discourse of the Human Sciences' (1980), while reminding us that his concept of deconstruction was developed in dialogue with structuralist thought, Derrida speaks of 'structure' rather than of conceptual systems, or of systems of thought. Even though it is not possible to discuss this point in depth here, I would like to point out how, in the context of that essay, 'structure' hints at as complex a formation as, for instance, the ensemble of concepts underlying social sciences, or even the whole of Western philosophy.

remain unthought within the conceptual structure of software?[8] In Derrida's words (1980), such a reading looks for a point of 'opacity', for a concept that escapes the foundations of the system in which it is nevertheless located and for which it remains unthinkable. It looks for a point where the conceptual system that constitutes software 'undoes itself'[9]. For this reason, a deconstructive reading of software is the opposite of a functional reading. For a computer professional, the point where the system 'undoes itself' is a malfunction, something that needs to be fixed. From the perspective of deconstruction, in turn, it is a point of revelation, in which the conceptual system underlying the software is clarified. Actually, I want to suggest that Derrida's point of 'opacity' is also simultaneously the locus where Stiegler's 'opacity' disappears, that is where technology allows us to see how it has been constituted. Being able to put into question at a fundamental level the premises on which a given conception of technology rests would prove particularly important when making decisions about it, and would expand our capacity for thinking and using technology politically, not just instrumentally.

Let me consider briefly some of the consequences that this examination of software might have for the way in which media and cultural studies deals with new technologies. We have already seen that the issue of technology has been present in media and cultural studies from the very beginning, and that the debate around technology has contributed to defining the methodological orientation of the field. For this reason, it is quite understandable that rethinking technology would entail a rethinking of media and cultural studies' distinctive features and boundaries. A deconstructive reading of software will enable us to do more than just uncover the conceptual presuppositions that preside over the constitution of software itself. In

[8] I am making an assumption here - namely that software is a conceptual system as much as it is a form of writing and a material object. In fact, the investigation of these multiple modes of existence of software is precisely what is at stake in my dissertation. In the context of the present introduction, and for the sake of clarity, I am concentrating on the effects of a deconstructive reading of a 'structure' understood in quite an abstract sense.

[9] According to Derrida, deconstruction is not a methodology, in the sense that it is not a set of immutable rules that can be applied to any object of analysis – because the very concepts of 'rule', of 'object' and of 'subject' of analysis, themselves belong to a conceptual system (broadly speaking, they belong to the Western tradition of thought), and therefore are subject to deconstruction too. As a result, 'deconstruction' is something that 'happens' within a conceptual system, rather than a methodology. It can be said that any conceptual system is always in deconstruction, because it unavoidably reaches a point where it unties or disassembles its own presuppositions. On the other hand, since it is perfectly possible to remain oblivious to the permanent occurrence of deconstruction, there is a need for us to actively 'perform' it, that is to make its permanent occurrence visible. In this sense deconstruction is also a productive, creative process.

fact, such an investigation will have a much larger influence on our way of conceptualising what counts as 'academic knowledge'. To understand this point better, not only must one be reminded that new technologies change the form of academic knowledge through new practices of scholarly communication and publication as well as shifting its focus, so that that the study of new technologies has eventually become a 'legitimate' area of academic research. Furthermore, as Gary Hall (2002: 111) points out, new technologies change the very nature and content of academic knowledge. In a famous passage, Jacques Derrida wondered about the influence of specific technologies of communication (such as print media and postal services) on the field of psychoanalysis by asking 'what if Freud had had e-mail?' (Derrida 1996). If we acknowledge that available technology has a formative influence on the construction of knowledge, then a reflection on new technologies implies a reflection on the nature of academic knowledge itself. But, as Hall maintains, paradoxically 'we cannot rely merely on the modern "disciplinary" methods and frameworks of knowledge in order to think and interpret the transformative effect new technology is having on our culture, since it is precisely these methods and frameworks that new technology requires us to rethink' (Hall 2002: 128). According to Hall, cultural studies is the ideal starting point for a study of new technologies, precisely because of its open and unfixed identity as a field. A critical attitude toward the concept of disciplinarity has characterized cultural studies from the start. Such a critical attitude informs cultural studies' own disciplinarity, its own academic institutionalisation (115).[10] Yet Hall argues that cultural studies has not always been up to such self-critique, since very often it has limited itself to an 'interdisciplinarity' attitude understood only as an incorporation of heterogeneous elements from various disciplines - what has been called the 'pick'n'mix' approach of cultural studies -but not as a thorough questioning of the structure of disciplinarity itself. He therefore suggests that cultural studies should pursue a deeper self-reflexivity, in order to keep its own disciplinarity and commitment open. This self-reflexivity would be enabled by the establishment of a productive relationship between cultural studies and deconstruction. The latter is understood here, first of all, as a problematizing reading that would permanently

[10] For the scope of the present Introduction, I assume Hall's term 'cultural studies' as roughly equivalent to what I have previously named 'media and cultural studies', since this passage refers to a constitutive debate around the field's conceptual framework.

question some of the fundamental premises of cultural studies itself. Thus, cultural studies would remain acutely aware of the influence that the university, as a political and institutional structure, exercises on the production of knowledge (namely, by constituting and regulating the competences and practices of cultural studies practitioners). It is precisely in this awareness, according to Hall, that the political significance of cultural studies resides. Given that media and cultural studies is a field which is particularly attentive to the influences of the academic institution on knowledge production, and considering the central role played by technology in the constitution of media and cultural studies, as well as its potential to change the whole framework of this (already self-reflexive) disciplinary field, I want to argue here that a rethinking of technology based upon a deconstructive reading of software needs to entail a reflection on the theoretical premises of the methods and frameworks of academic knowledge.

To conclude, in this thesis I propose a reconceptualization of new technologies, that is of technologies based on software, through a close, even intimate, engagement with software itself, rather than through an analysis of how new technologies are produced, consumed, represented and talked about. To what extent and in what way this intimacy can be achieved and how software can be made available for examination are the main research problems of this thesis. Taking into account possible difficulties resulting from the working of mediation in our engagement with technology as well as technology's opacity and its constitutive, if unacknowledged, role in the formation of both the human and academic knowledge, I want to argue via close readings of selected software practices, inscriptions and events that such an engagement is essential if we are to take new technologies seriously, and to think them in a way that affects - and that does not separate – cultural understanding and political practice.

1 From Technical Tools to Originary Technicity

The Concept of Technology in Western Philosophy

In this chapter I suggest that the problem of 'new technologies', and of the kind of knowledge that can be produced about them, cannot be addressed without radically reconsidering what we mean by 'knowledge' in relation to 'technology' in a broader sense. What kind of knowledge can we produce about technology in the context of media and cultural studies, apart from the analysis of the discourses and practices of producers and consumers? To answer this question, I argue that, as a preliminary step, the received concepts of technology need to be put into question. By received concepts of technology I mean the ways in which technology has been understood primarily by the Western philosophical tradition.

This turn to the philosophical conceptions of technology in the context of media and cultural studies might seem somewhat daring or even misjudged. However, I argue that media and cultural studies can highly benefit from a productive dialogue with philosophy on the subject of technology. Although a detailed discussion of the acceptability of interweaving philosophy and media and cultural studies would take the present chapter too far, it must be noticed that a debate on the relevance of philosophical thought has taken place , from time to time, throughout the history of media and cultural studies. This debate has mainly focused on 'theory', that is on specific developments in French structuralist and post-structuralist thought, such as semiotics and deconstruction. Nevertheless, I argue that this capacity for questioning its own conceptual framework is precisely what enables media and

cultural studies to *think technology* originally and innovatively, and therefore to interrogate what we mean by technology in the first place.

To give but one example, in his famous essay 'Cultural Studies and Its Theoretical Legacies', Stuart Hall takes into consideration the tension between theoretical and political dimensions that for him determines the specificity of cultural studies:

> Both in the British and the American context, cultural studies has drawn the attention itself, not just because of sometimes dazzling internal theoretical development, but because it holds theoretical and political questions in an ever irresolvable but permanent tension. It constantly allows one to irritate, bother, and disturb the other, without insisting on some final theoretical closure.
>
> (Hall 1992: 284)

According to Hall, the theoretical encounters with structuralism and post-structuralism have forced cultural studies to constantly question itself and to keep its identity open and heterogeneous. And yet what holds the field together is its politically committed nature. 'Not that there is one politics already inscribed within it. But there is something *at stake* in cultural studies in a way that I think, and hope, is not exactly true of many other very important intellectual and critical practices' (Hall 1992: 278).[1]

In his book of 2002 entitled *Culture in Bits*, Gary Hall observes that, while acknowledging the tension between theory and politics, Stuart Hall is actually inclined to give priority to the latter: therefore politics remains that which limits the destabilizing and decentering effects of theory (Hall 2002). Gary Hall takes a much more far-reaching stance by suggesting that theory itself has political relevance in cultural studies. For him, by enabling reflexivity within cultural studies, theory also

[1] For the scope of the present chapter, I take the term 'cultural studies' as roughly equivalent to what I have previously named 'media and cultural studies', since this passage refers to a fundamental discussion about the field's methodology and conceptual framework. Also, in the UK academic context, the discipline of cultural studies has morphed into a broader cross-disciplinary field of 'media, communications and cultural studies' over the recent years.

enables cultural studies to become particularly aware of the influences that the university as a political and institutional structure exerts on the production of knowledge, including knowledge produced within cultural studies. Thinking politically means first of all being attentive to the institutional forces that shape thought itself, such as the constitution and regulation of cultural studies practitioners' competences by the university.

Therefore, since the ability to question inherited conceptual frameworks appears to be one of cultural studies' points of strength, and since this dissertation aims at 'demystifying' new technologies and developing new forms of knowledge about technology within media and cultural studies, I want to argue here that a re-examination of the philosophical conceptions of technology is a convenient starting point for my argument.

Let me thus begin with a general proposition: Western philosophy has always found it rather difficult to think about technology. For instance, in the first volume of his book *Technics and Time* (1998), Bernard Stiegler remarks that, while the extraordinary technological changes of our age need to be conceptualized and made intelligible as soon as possible, in attempting to achieve this intelligibility one cannot rely on any available account of technology in the Western philosophical tradition: 'at its very origin and up until now, philosophy has repressed technics as an object of thought. Technics is the unthought' (Stiegler 1998a: ix).[2]

Although later on in his work Stiegler identifies a few exceptions to this philosophical refusal to openly approach technology – namely, the thought of several French philosophers, including Jacques Derrida, and that of Martin Heidegger - he nevertheless points out that philosophical reflection has traditionally pushed technology to its own margins. And yet, a critical evaluation of such reflection shows how the concept of technology has always been tightly connected to the concepts of 'knowledge', 'language' and 'humanity'.

[2] The term 'technics' belongs to Stiegler's partially Heideggerian philosophical vocabulary. I take it here as a synonym for what we commonly refer to as 'technology'. I will point out specific uses of the term where appropriate.

For this reason, in the present chapter I take into consideration a number of philosophers who have attempted a conceptualisation of technology and dealt with the difficulty of producing knowledge about it. I examine both the dominant philosophical conception of technology based on the Aristotelian thought, which substantially reduces technology to a mere instrument, and the work of those thinkers who have distanced themselves from such an instrumental understanding and have instead proposed a view of technology as a fundamental characteristic of human beings. I refer mainly, but not exclusively, to the work of Heidegger, Stiegler, Derrida and the French palaeontologist André Leroi-Gourhan. The work of all these thinkers shows that philosophy has constituted itself precisely in relation (and in opposition) to technological knowledge, and therefore it points to the need for the radical rethinking of philosophy itself if an understanding of technology is to be made possible.

Tracing a map of the philosophical thought on technology is not an easy task. In order to start exploring this problem, let me initially follow the innovative genealogy proposed by Stiegler (1998a). Stiegler's position on the relationship between philosophy and technology is quite striking. Although, as we have seen above, he argues for the 'urgency and necessity of an encounter between philosophy and technology' (Stiegler 1998a: xi), he actually views philosophy as traditionally and constitutively incapable of thinking technology.

Stiegler identifies an interesting paradox in the contemporary way of understanding technology: while on the one hand we keep conceptualizing technology in the traditional terms of means and ends, on the other hand we are faced today by a new technological 'opacity' (14). This opacity makes apparent the inadequacy of our conceptual frameworks to understand contemporary technology. Furthermore, Stiegler attributes this inadequacy to today's 'breakdown of knowledge into separate domains' (14). He emphasizes the insufficiency of the so-called 'inter-disciplinary' approach to the study of technology. In fact, he claims that today's technology calls for a profound rethinking of the relationship between technology itself and culture.[3]

[3] Quite similarly, as I have pointed out in the Introduction, Gary Hall (2002) argues that 'new technologies' stimulate a rethinking of the concept of disciplinarity.

Stiegler attributes the problematic character of today's technology to its relation with time. In other words, contemporary technology is problematic because the pace of technological innovation has become very fast. New technologies emerge and rapidly make obsolete more and more pre-existing technologies, as well as the social and cultural practices associated with them. He writes:

> Innovation is inevitably accompanied by the obsolescence of existing technologies that have been superseded and the out-of-datedness of social situations that these technologies made possible – men, domains of activity, professions, forms of knowledge, heritage of all kinds that must either adapt or disappear.
>
> (Stiegler 1998a: 14)

I want to point out here how the experience that Stiegler describes in this passage is all too familiar to anyone who has paid attention to the rapid succession of 'generations' of personal computers or mobile phones. Besides, the anxiety of keeping up with innovation is particularly felt by computer professionals and other practitioners of technological fields, who need to constantly 'update' and refresh their competencies just to be able to keep their jobs.

The problem of obsolescence, of out-of-datedness, is key to our uncomfortable relationship with technology today, Stiegler claims. Therefore, he identifies an apparent divorce between technical and scientific knowledge on the one hand, and culture on the other. Hence the urgency of needing to rethink the modalities of the interaction between technology and culture.[4]

[4] A similar argument on technology and time has also been made by the French philosopher and media theorist Paul Virilio. In his theory of the 'visual machine' Virilio focuses on the 'automation of vision' – that is, the transferring to computers of perceptual functions traditionally bound up with the body. He emphasises that, in contrast with earlier visual technologies (such as the telescope, the microscope, or even cinema), which used to extend the perceptual capacities of the body, contemporary vision machines bypass the constitutive limits of our body thanks to their ultrahigh-speed operations. The ultra-fast machine vision neglects the longer time of exposure needed by the human ocular system to memorize an image (Virilio 1994: 61). An original interpretation of Virilio's theory is offered by Mark Hansen, according to whom Virilio, rather than simply showing how machines are making the human superfluous, actually argues for the right of humans to see 'differently' from machines (Hansen 2004; Virilio 1997).

To develop this point further, Stiegler draws on Bertrand Gille's concept of 'permanent innovation' (in which new technologies continually spread in society) as the basis of industrial civilization (Gille 1986; Stiegler 1998a). This process of permanent innovation results in a difference in speed between technical and cultural change. Moreover, according to Stiegler, contemporary technology has a totally new relation with time. He expresses this fact with the image of a technological device that 'goes faster than its own time'. Stiegler's favoured analogy is that of 'a supersonic device, quicker than its own sound', whose breaking of the sound barrier provokes 'a violent sonic boom, a sound shock' (Stiegler 1998: 15).

Stiegler indicates two instances of this transformed relationship between technology and time: what goes under the name of 'live' media, and what we call 'real-time' computers. I do agree with Stiegler's claim that both 'live' media and 'real-time' computers deeply modify (or, in his terms, 'distort') the taking place of time (what he calls 'event-ization', *événementialisation*). Yet, it must be noted in passing that here Stiegler overlooks the substantial difference between these two kinds of technology. In fact, while 'live' media give us access to an event 'without delay', the concept of 'real-time' defines the capacity of a computer to respond to changes in its environment 'as soon as they happen' - that is, in a fast and effective way. Since these changes may occur at a speed of milliseconds, the time of response belongs to an order of temporality practically unperceivable by human beings. Therefore, the concept of 'live' media seems to retain a 'human' measure of time that 'real-time' computers have renounced. At the same time, the latter present a level of risk in a certain way ignored by 'live' media (real-time systems are technically defined as 'systems for which time is critical', and they are typically involved in controlling complex apparatuses in potentially dangerous situations, such as airplanes during flight).[5]

However, the profound involvement of technology with time becomes apparent today through both the speed of technical change and the ruptures in event-ization that this change provokes. In Stiegler's words, 'there is today a conjunction between

[5] Again, a similar argument about the capacity of real-time technologies to bypass human perceptions has been made by Virilio (1994, 1997).

the question of technics and the question of time' that 'calls for a new consideration of technicity' (17).

Before examining Stiegler's chart of the philosophical thought on technology further, let me point out how his emphasis on the particularly fast pace of contemporary technological change resonates with the problem of defining the 'newness' of 'new technologies' and 'new media' - that is, with the question of what actually is 'new' in 'new technologies' and 'new media', a problem that has been widely discussed in media and cultural studies.

For instance, in her 1988 book *When Old Technologies Were New*, media theorist Carolyn Marvin shows how the process of electrification, which started at the end of the nineteenth century and led to the electrical technologies in use today, was perceived by its contemporaries as extremely upsetting and challenging – in a word, 'new' (Marvin 1988). The deliberate anachronism of Marvin's title indicates that the technologies we now take for granted (to the point that they have become almost invisible to us) were once regarded as tremendously innovative, and that the process that led to their adoption involved a great deal of complex social and cultural adjustment.

Albeit Marvin conceives her study as an historical monograph focusing on the dynamics of the social reception of electricity, and although she does not propose any explicit analogy with contemporary technology, her work effectively destabilizes the rhetoric of 'newness' that can be found in contemporary discourses around 'new technologies' (see, for instance, Lister et al. 2003). However, it seems to me that Marvin's analysis remains at the level of the social and cultural reception of technological innovation, while Stiegler attempts to question the concept of innovation itself, as well as the relationship between what we identify as 'technology' and 'culture'.

In sum, according to Stiegler the speed of contemporary technology makes it particularly difficult to understand it. His observation stimulates the following questions: in what way can we think the relationship between contemporary technology and time? Is this relationship distinctive of 'new technologies', or does

its relevance extend to *all* technology? How can we understand this relation more generally, and in what way would such an understanding change our concept of technology - and possibly even of time? In other words, Stiegler's argument opens up two orders of questions, one concerning the 'newness' of the relationship between 'new technologies' and time, and the other regarding ways of framing the relationship between technology and time in general.

According to Stiegler, the first philosopher to seriously think technology in relation with time was Martin Heidegger. He was also the first to address some of the reasons for philosophy's incapacity to think technology. To understand this point better, and before examining Heidegger's argument on technology, let me spend a little more time on Stiegler's understanding of the relationship between technology and philosophy.

For Stiegler philosophy has always 'repressed' technology as an object of thought. What Stiegler means by this is that, from the very beginning, Western philosophy has distinguished itself from technology, and has in fact identified itself as *not* technology. It has done so by separating *technê* from *epistêmê. Epistêmê* is the Greek word most often translated as knowledge, while *technê* is translated as either craft or art (Parry 2003). The separation between *technê* and *epistêmê* was rooted in the political arena of fifth century Athens, and it associated *technê* with the rhetorical skills of the Sophists. As professional rhetoricians, the Sophists were skilled in the construction of political arguments. Their skillfulness (*technê*) was perceived as indifference to establishing truth, or, worse, as an attempt to make truth instrumental to power. As such, Sophists' *technê* came to be opposed to true knowledge. Therefore, truth remained the only object of *epistêmê,* which in turn was identified with philosophy. This substantially political move deprived technical knowledge of any value.

Stiegler emphasizes that the subsequent step in the devaluation of technology was made by Aristotle through his definition of a 'technical being' as something that does not have an own end in itself and that is just a tool used by someone else for

their ends.[6] It must be noted here that Stiegler leaves out the fact that Aristotle also specifically addresses the distinction between *technê* and *epistêmê* in a few passages of his work. As Richard Parry points out, the most obvious place to begin an examination of *epistêmê* and *technê* in Aristotle's writings is Book VI of the *Nicomachean Ethics*, where Aristotle makes a clear distinction between the spheres of scientific knowledge (*epistêmê*) and craft (*technê*). Parry explains:

> Scientific knowledge concerns itself with the world of necessary truths, which stands apart from the world of everyday contingencies, the province of craft. Although there are a few problems of interpretation surrounding this description of scientific knowledge (mainly because Aristotle is quite ambivalent in his use of the term *epistêmê* in *Posterior Analytics* and in *Metaphysics*), we have here a classic distinction between the "purely theoretical" and the "purely practical".
>
> (Parry 2003: non-pag.)

However, this further analysis of Aristotle's thought does nothing but reinforce Stiegler's argument that the exclusion of technology from philosophy has been founded on the concept of instrumentality: technical knowledge has been interpreted as instrumental, and therefore as non-philosophy. Timothy Clark synthetically clarifies 'the conception of technology that ... has dominated Western thought for almost three thousand years' (Clark 2000) as follows:

> The traditional, Aristotelian view is that technology is extrinsic to human nature as a tool which is used to bring about certain ends. Technology is applied science, an instrument of knowledge. The inverse of this conception, now commonly heard, is that the instrument has taken control of its maker, the creation control of its creator (Frankenstein's monster).
>
> (Clark 2000: 238)

[6] *Nicomachean Ethics* 6, 3-4 (Aristotle 1984).

Clark's passage shows quite clearly the Aristotelian basis of the utilitarian model of technology which is still in use today.

Moreover instrumentality has gained a new importance during the process of the industrialization of the Western world. Accordingly, technology has slowly acquired a new place in philosophical thought. Stiegler maintains that science has become more and more instrumental (to economy, to war) in the course of the last two centuries, therefore gradually renouncing its character of 'pure' knowledge. At the same time, philosophy has become interested in the 'technicization' of science. As an example of this Stiegler cites Edmund Husserl's work on the arithmeticization of geometry.

During the ascent of Nazism in Germany, Husserl conceptualized the emergence of algebra (which had been ongoing since Galileo's times) as a technique of calculation that emptied geometry of its visual content. 'In algebraic calculation,' he wrote, 'one lets geometric signification recede into the background as a matter of course, indeed one drops it altogether; one calculates, remembering only at the end that the numbers signify magnitudes' (Husserl 1970: 44-45). According to Husserl, by becoming viable to calculation, geometry renounces its capacity of visualizing geometrical shapes – or, in Husserl's terms, 'spatio-temporal idealities' (41). Therefore, as Stiegler comments, 'the technicization of science constitutes its eidetic *blinding*' (Stiegler 1998a: 3).[7]

I want to point out here how the concept of 'calculation' is a constitutive part of the concept of instrumentality. For Husserl calculation seems to be the equivalent of formalization and algebra, as a technique of calculation, is nothing but a formalism that allows us to manipulate numerical configurations and to forget their visual meaning. The emphasis here is not on a supposedly 'mechanical' character of calculation; on the contrary, Husserl highlights the fact that algebra still makes geometrical discoveries possible. Rather, the emphasis is on the forgetting of what Husserl understands as the visual meaning of geometry.

[7] The term 'eidetic' comes from the Greek *eidos*, which means form. Therefore, here Stiegler hints at a loss of the 'visual' dimension of geometry.

Stiegler also points out how the Platonic conception of technicization as the loss of memory is still at the basis of Husserl's understanding of algebra (3). I will come back to Plato's understanding of technology later on in this chapter, particularly when addressing Derrida's conception of writing. For now it is worth remembering that in his dialogue *Phaedrus* Plato famously associates writing, understood as a technique to aid memory, with the loss of true memory, which for Plato is *anamnesis*, or recollection of an ideal truth. From this perspective, which again separates knowledge from technology, writing is devalued because of its instrumentality.[8]

To recapitulate Stiegler's argument so far, the devaluation of technology in Western philosophy goes hand in hand with the devaluation of writing. Thus, Stiegler establishes an important relationship between technology and writing as both excluded by knowledge and encompassed by the concept of instrumentality. In the context of my investigation of new technologies Stiegler's argument seems to imply that the question to be asked is not just whether an instrumental concept of technology is adequate for an understanding of new technologies. Another question needs to be posed at this point – namely, if new technologies exceed and destabilize the concept of instrumentality, do they not also destabilize the concept of writing as instrumental? And what would the consequences of such a destabilization be? I will return to these two questions later on in this chapter and in the following one. For now let me investigate Stiegler's reconstruction of the philosophical tradition of thought on technology a little further.

According to Stiegler, it is Heiddeger's understanding of technology that offers the first opportunity to rethink instrumentality and consequently the relationship between technology and knowledge. Famously, Heidegger saw technology as responsible for 'the spiritual decline of the earth'. Nonetheless – as Stiegler emphasizes – he was also the first philosopher to seriously think technology after Marx. Mark Poster has pointed out that Heidegger's antipathy towards technology was accompanied by an enormous sensitivity to the problem of technology itself, and that he was 'no simple technophobe' (Poster 2002: 17). Poster's rereading of

[8] *Phaedrus* 275 ff. (Plato 1989).

Heidegger's influential 1955 essay, 'The Question Concerning Technology', is particularly helpful since he seeks to explicitly test the validity of Heidegger's thought for an understanding of new technologies. In fact, Poster claims that Heidegger's thought is based on a homogeneous, 'unified' idea of technology that cannot give full account of actual, specific technologies, and that cannot thus be applied to information technologies.

Poster shows how the central point of Heidegger's argument in this essay is not technology *per se* but modern humanity's way of being. Technology characterizes modern 'culture' – the term that Poster chooses for Heidegger's *Dasein* (Poster 2002: 18). For Heidegger, humanity has to 'bring itself forth' in order to be, and it does so in part through its use of things - that is, through technology. In this context, technology is understood as a whole process of setting up the world. As long as humanity is aware of this process, it has a free relation to itself. This was the case in ancient Greece, where technology was openly visible and integrated into culture. However, in the modern age, technology has become a way of using things which brings humanity forth, while at the same time concealing this very process - that is, concealing technology itself. Ultimately, modern technology 'challenges' nature: it does violence to it and reduces it to an available resource. In so doing, it also reduces humanity to the same status, since humanity as part of nature becomes a servant of technology. Heidegger calls this process 'enframing'. His hope is that humankind recognises this process of enframing and becomes capable of developing a kind of technology which would be completely different from today's (Poster 2002: 18).

This brief, even schematic, synthesis of Heidegger's thought nevertheless allows us to see that he does not view technology as essentially instrumental. On the contrary, technology is for him a way of being in the world. This is the sense of his famous affirmation that 'the essence of technics is nothing technical' (Heidegger 1977: 35). According to Stiegler, this is precisely what makes Heidegger's understanding of technology so interesting. Heidegger suggests that, if we keep thinking technology as a 'means', we will never be able to understand what technology *is*. In other words, we cannot think technology efficaciously as long as we remain in the frame of mind of instrumentality.

Stiegler also emphasizes that Heidegger's understanding of technology is deeply connected to his conception of time. For Heidegger, calculation has its roots in our relation to the future, and in our attempt to determine future possibilities, which we fear precisely because they appear indeterminate. Heidegger describes this process as 'anticipation' or 'concern': our attempt to control (or to anticipate) the uncertainty of the future creates the basis for calculation, or for circumscribing the realm of possible futures. Understood in a broader historical context, this is what Heidegger identifies as the turning of Western thought into calculation in the modern age. This is also why for Heidegger technology has a central role in defining modernity. As Stiegler comments, 'the *modern* age is essentially that of modern *technics*' (Stiegler, 1998a: 7). On the other hand, according to Heidegger, modern technology also opens up for us the possibility of radically reconceiving technology itself by becoming conscious of the instrumental approach which has characterized our understanding of technology since Aristotle. This is why Stiegler praises Heidegger as the first philosopher who dares to propose a radical reconceptualization of technology.

Importantly, Heidegger tracks the instrumental conception of technology back to the classical theory of the four causes, as elucidated in Aristotle's *Nicomachean Ethics* and *Physics* (Heidegger 1977: 7). Simply put, in the Aristotelian perspective technology is seen as instrumental because it is conceptualized in relation to ends and means. The producer of the technical object (i.e. a tool) acts as its 'efficient cause'. But the technical object does not have its 'final cause' (that is, its end) in itself – and the reason for this is again to be found in the artificiality of the technical object: only natural beings are understood to have an end in themselves. The end of the technical object belongs to the producer, and precisely for this reason the technical object is nothing more than a means. For Heidegger, *technê* as production (what the Greeks called *poiesis*) brings into being that which did not exist. In Heidegger's own terms, *technê* 'discloses' it (13). It is worth noting that for Heidegger ends are themselves part of this disclosure. Thus, as we have seen, the main sense of *technê* has nothing to do with manipulating, making or using tools, but rather signifies 'revealing'. And yet modern technology treats nature as a supplier of energy that must be extracted and stored. In this process, both nature and

humanity are put under the imperatives of technology. Technology becomes a project of calculation intended to master nature and humanity, and this is what – as we have seen earlier - Heidegger describes as the 'enframing' of nature and humanity through calculation (19).

Stiegler also identifies a 'Marxist offshoot' to Heidegger's thought -- developed for instance by Jürgen Habermas - which nonetheless did not manage to escape an instrumental conception of technology. Let me now discuss Stiegler's rereading of Habermas' work briefly in order to show how, on the one hand, Habermas' theory of technology, albeit extremely interesting, does not account for the way in which contemporary technology works, and therefore does not contribute to its demystification and, on the other hand, why Heidegger's position seems to me to hold a better promise for an understanding of 'new' technologies. Unlike Heidegger, Habermas does not propose a radical reconceptualization of the philosophical conception of technology. He identifies in modern society a form of technocracy, that is of political domination which is not recognizable as such because it derives its legitimation not from the political arena but rather from scientific and technical knowledge. More precisely, the nature of technocracy is to confuse political legitimation and techno-scientific legitimation, therefore making techno-scientific knowledge the only visible source of legitimacy, and transforming any opportunity for political debate into a malfunction in a society that needs to be fixed. Technocracy substantially depoliticizes society. Furthermore, language itself is 'technicized', since technical and scientific frames of mind spread all over society and include also communication. Habermas and Heidegger both conceptualize technical modernity as a paradoxical situation in which technology ends up doing a disservice to humanity rather than being in its service. Nevertheless, Habermas continues to analyze technology in terms of ends and means. He suggests that we pursue a liberation of language from its technicization, and that we turn technology into an object of democratic debate in a free language. On the contrary, Heidegger problematizes the very concept of 'means' and, much more radically, suggests that we rethink 'the bond originarily formed by, and between, humanity, technics, and language' (Stiegler 1998a: 13).

From the above formulation, it appears clear that for Habermas technology can be treated as an object of a discussion that takes place in a transparent language and that is based on what he calls 'good reasons', that is rationally convincing arguments. He seems to believe that technology does not have any real effect on language itself, or at the very least that the language of politics can be separated from the language of techno-science. Furthermore, he does not really take into account the transformation that information and communication technologies introduce in the 'public sphere', which is supposed to be the space of a democratic debate (Habermas 1989, 1991).

In order to make it even clearer why I consider Heidegger's position more promising than Habermas', let me now leave Stiegler for a moment and examine Habermas' more recent work on technology. In his book of 2001, entitled *The Postnational Constellation*, Habermas strives to understand the nature of information and communication technologies. Although he generally views contemporary technology in continuity with the past, he also recognizes that information processing, the Internet, and in general the speed of today's communication give us a 'new' experience of space and time. Thus, on the one hand, contemporary technology still runs 'along familiar lines': '[s]ince the seventeenth century, the instrumental attitude toward a scientifically objectified nature has not changed; nor has the manner in which we control natural processes, even if our interventions into matter are deeper, and our ventures into space are further, than ever before' (Habermas, 2001: 41). On the other hand, in another (albeit quite marginal) passage, Habermas acknowledges that the acceleration of communication processes has an impact on our everyday life:

> The acceleration effects of improved transport and communication technologies have an entirely different relevance for the long-term transformation of everyday experience. ... Digital communication finally surpasses all other media in scope and capacity. More people have quicker access to greater volume of information, and are able to process it and instantly exchange it over any distance. The mental consequences of the Internet ... are still very hard to assess.
>
> (Habermas 2001: 41)

Nevertheless, it is not until his later work on biotechnologies that Habermas develops an argument for the radical novelty of contemporary technology . Yet, even there, his understanding of technology does not seem to escape the general framework of instrumentality. In *The Future of Human Nature* (2003) Habermas claims that biotechnologies generate unprecedented moral problems and that genetic technologies are capable of affecting what it means to be human. To give but a synthesis of Habermas' position, enhancement-oriented genetic practices seem to him to entail an asymmetrical relationship of influence between generations, since the genetic programming of a child threatens his future freedom when it comes to choosing and shaping his own destiny. The collective deliberative process that seems to be Habermas's favoured solution when dealing with both technology and politics does not function here because future generations cannot take part in it. Habermas' attempt to solve this problem consists in proposing the concept of 'species ethics'- that is, the domain of decisions made by the human species as a whole about the question of what it means to be human. Our concern for ensuring future persons' status as free and equal beings is situated by Habermas at the level of species ethics. Therefore, although Habermas understands genetic technology as deeply troubling the very idea of humanity, he ultimately reinforces the latter by simply shifting the traditional values of individuality, equality and freedom to the level of the species. What it means to be human remains unquestioned, and so do the possible political consequences of such questioning.

I have briefly examined Habermas' position here in order to reiterate my earlier point that a deeper understanding of technology and of its relation with the human is needed if we are to understand the political implications of new technologies. In other words, we need to rethink technology philosophically if we want to think it politically. The question is not one of discussing technology 'democratically' through a 'freed' language. It is rather a matter of recognizing the mutually constitutive implications of technology and language – possibly via the concept of instrumentality – and therefore of radically rethinking both terms *together*, since there is no way of (re)thinking one without the other.

In order to develop this point, I am going to continue with my examination of the alternative tradition of thought on technology that, according to Stiegler, starts with Heidegger and is not based on the concept of instrumentality. Clark (2000) calls this the tradition of 'originary technicity' – a term he borrows from Richard Beardsworth (1996). This term assumes a paradoxical character only if one remains situated within the instrumental conceptualisation of technology: if technology were instrumental, it could not be originary – that is, constitutive of the human. Therefore, the concept of 'originary technicity' resists the utilitarian conception of technology. To clarify what he means by 'originary technicity', Clark refers to the 1992 novel, *The Turing Option*, co-authored by Marvin Minsky, a leading theorist in the field of Artificial Intelligence (Harry and Minsky 1992).[9] In order to regain his cognitive capacities after a shooting accident has severely damaged his brain, the protagonist of the novel, Brian Delaney, has a small computer implanted into his skull as a prosthesis. After the surgery he starts reconstructing the knowledge he had before the shooting. The novel shows him trying to catch up with himself through his former notes and getting an intense feeling that the self that wrote those notes in the past is lost forever. Clark uses this story as a brilliant figuration of the fact that no self-consciousness can be reached without technology:

> Delaney's experience in *The Turing Option* is only different in degree from the normal working of the mind from minute to minute ... No thinking – no interiority of the psyche – can be conceived apart from technics in the guise of systems of signs which it may seem to employ but which are a condition of its own identity.
>
> (Clark 2000: 240)

This passage shows that Clark understands 'technics' not in terms of massive engineering works but as 'the subtler intimacy of the relation of technology to human thinking', and especially as 'the intimacy between technology and language'

[9] Artificial Intelligence (AI) is a research area that aims at developing 'intelligent machines', that is, broadly speaking, computers endowed with cognitive capabilities equivalent to those of human beings. This definition of AI draws on the work of one of its founders, the English mathematician Alan Turing, who in 1950 devised a test to assess the presence of intelligence in computers. This test, called 'imitation game', basically established that a computer must be considered intelligent when it is capable of imitating a human being convincingly in terms of linguistic behaviour (Turing 1950; Nilsson 1998; Callan 2003; McCorduck 2004; Leavitt 2006).

(240). Such an understanding of technology ostensibly draws on Heidegger's thought, as well as on Derrida's and Stiegler's.

In order to clarify the concept of 'originary technicity' further and to investigate its significance for my analysis of new technologies, let me now return to Stiegler's work. What Clark (2000) calls 'originary technicity', Stiegler names 'originary prostheticity' of the human (Stiegler 1998a: 98-100). To understand the latter better, it is helpful to examine Stiegler's essay 'The Time of Cinema' and the third volume of *Technics and Time*, particularly where – in dialogue with Derrida - he reworks Husserl's philosophy of time (Stiegler 1998b, 2001a, 2003a).

Stiegler's philosophy of technology is based on the central premise that 'the human has always been technological' (Hansen 2003: non-pag.). Stiegler draws here on the work of the French paleontologist André Leroi-Gourhan, who tightly connects the appearance of the human with tool use. For Stiegler, too, the human co-emerges with tool use. He writes:

> Humans die but their histories remain – this is the big difference between mankind and other life forms. Among these traces most have in fact not been produced with a view to transmitting memories: a piece of pottery or a tool were not made to transmit any memory but they do so nevertheless, spontaneously. Which is why archaeologists are looking for them: they are often the only witnesses of the most ancient *episodes*. Other traces are specifically devoted to the transmission of memory, for example writing, photography, phonography and cinematography.
>
> (Stiegler 2003a: non-pag.)[10]

The above passage demonstrates that for Stiegler, technology carries the traces of past events. In Mark Hansen's words, it is 'the support for the inscription of memory' (Hansen 2003: non-pag.) – that is, technology is always a memory aid, and

[10] I quote here from the English translation of Chapter 4 of Stiegler's *La technique et le temps 3. Le temps du cinéma et la question du mal-être*, (2001a), published in *Culture Machine* (Stiegler 2003a). Stiegler's original intake on the correlation between the human and the technological informs his rereading of Heidegger, as well as his divergence from Derrida's own reworking of Heidegger (Hansen 2004).

only through memory do human beings gain access to their own past, and therefore become aware of themselves, or gain a consciousness.

This understanding of technology as inscribing the memory of the past is further illustrated by Stiegler in an interview included in the documentary *The Ister*, directed by David Barison and Daniel Ross in 2004.[11] Any technical instrument, Stiegler states in this interview, registers and transmits the memory of its use. For instance, a carved stone used as a knife preserves the act of cutting, thus becoming a support for memory. In this sense, technology is the condition of the constitution of our relation to the past.

In sum, according to Stiegler it can be said that human beings 'exteriorize' their memory into technological objects, which in turn are nothing but memory exteriorized. Importantly, by doing this the human species becomes able to suspend its genetic program and to evolve through means other than animal instincts - that is, in Stiegler's words, to 'pursue life through means other than life' (Stiegler 1998a: 17). Stiegler gives the name of 'epiphylogenesis' to this process, (Stiegler 2003a: non-pag). Epiphylogenesis is the transformation and evolution of the human species through its relationship with technology, rather than only on the basis of its genetic program.

Furthermore by functioning as a support for memory, a technical object for Stiegler 'forms the condition for the givenness of time in any concrete situation' (Hansen 2003: non-pag.). For this reason, Stiegler maintains that human beings can experience themselves only through technology.[12] This formulation becomes much clearer if we consider cinema, which for Stiegler is the emblematic technology of contemporaneity. Hansen (2003) comments:

---

[11] *The Ister* is a filmic philosophical meditation on Heidegger, the changing nature of European culture, and the role of technology and philosophy. While taking the viewer on a journey from the mouth of the Danube river (whose ancient name was Ister) in Romania to its source in the Black Forest, the film incorporates interviews with Bernard Stiegler, Jean-Luc Nancy, Philippe Lacoue-Labarthe and the German filmmaker Hans-Jürgen Syberberg.

[12] Such an experience of the self is what philosophers have called 'self-affection' (Kant) or - and this is particularly important in Stiegler's thought, as I will show in a moment - 'internal time-consciousness' (Husserl).

> More than any other technology (and certainly more than literature), it is cinema in its contemporary form as global television that frames time for us and gives us a surrogate temporal object in whose reflection we become privy to the flux of our own consciousness. At the same time, by opening consciousness onto the past, onto the non-lived tradition of historicality, onto otherness of that which does not belong to the experience of consciousness, cinema *qua* temporal object captures the contemporary manifestation of the interdependence of the who and the what, of the human subject and the technical other. Put bluntly, we become who we are by inheriting a past destined to us through cinema.
>
> (Hansen 2003: non-pag.)

As Hansen explains in the above passage, for Stiegler cinema (and, by extension, technology) makes available to us the experience of others, and therefore constitutes a striking example of the relation between technological objects and time. This complex passage is partly based on a Husserlian terminology, and on Stiegler's rereading of Husserl's phenomenological thought, which in turn constitutes the basis of Stiegler's analysis of the relation between technology and time. Before addressing Stiegler's reworking of Husserl's thought in detail, it is worth noting that the above passage opens up a whole new series of questions in relation to my investigation of new technologies. Firstly, by analogy, one could ask: what kind of temporality becomes accessible to us through new technologies? Or, to be more precise, and since - as I have explained in my Introduction - for the scope of the present dissertation I take 'new technologies' as a synonym for 'software-based technologies', what kind of temporality becomes accessible to us through software-based technologies and, specifically, through software?[13] As I have shown earlier on in this chapter, software-based technologies, such as real-time technologies, can, on the one hand, bypass the human perception of time. On the other hand, different

[13] As I clarified in my Introduction, albeit not every aspect of new technologies can be reduced to software, I maintain that digital codes and languages – known as 'software' as a whole – are central for the functioning of new technologies. However, I do not presume to know in advance what 'software' *is* and I do not take its accessibility for granted. In fact, I will investigate the different meanings of the term 'software', as well as the possibility and modes of engaging with it, in the next chapter and through the whole course of my dissertation.

kinds of software-based technologies, such as word processors, operate on a much more 'human' scale of temporality. Moreover, common to all software-based technologies is the fact that, before they become operative, they must be programmed – that is, software must be designed and developed. Therefore, one could ask a second question: what kind of temporality do we access through programming and what kind of relation to ourselves do we establish through software? Here I want to argue that Stiegler's rereading of Husserl's phenomenology can help with answering these questions.

As Hansen explains in his careful analysis of Stiegler's thought, the experience of others that we have not directly experienced but that becomes accessible to us because it has been recorded is what Stiegler calls 'tertiary memory' (Hansen 2004: 254). Stiegler draws here on Husserl's concept of 'image-consciousness'. An example of image-consciousness is for Husserl a painting 'where the artist ... archives her experience in the form of a memory trace' (316). This trace is an image of the past and of the memory of the artist, but it is not an image of the lived past of the viewer. While for this reason Husserl excludes image-consciousness from any role in time-consciousness, Stiegler reverses the argument: for him tertiary memory (that is, memory that has not been lived through by us) is the very condition of time-consciousness. In other words, Stiegler foregrounds a consequence of Husserl's thought that Husserl himself hesitated to recognize: namely, the intrinsically technical basis of our consciousness of time.

To be more specific, Husserl recognizes that we cannot grasp temporality by a direct analysis of consciousness, and that we necessarily need to examine 'an object that is itself temporal'. A temporal object is defined as 'an object that is not simply in time but is constituted through time and whose properly objective flux coincides with the flux of consciousness when it is experienced by a consciousness. Husserl's favoured example is a musical melody' (Hansen 2004: 254). As Hansen points out, by

focusing on the temporal object Stiegler can complicate Husserl's analysis of time-consciousness and introduce technology into it.[14]

However, what is important here is that Stiegler reverses Husserl's hierarchy. We do not have a primary understanding of time and *then* technology: it is rather technology that gives us an understanding of time. The reason for this is that we always find ourselves in the midst of a horizon – a world already constituted by and comprising both what we had experienced in the past *and* the past we never experienced (but that was experienced by others and given to us through technical memory supports).[15]

To recapitulate Stiegler's argument, the relationship between time and technology is for him a fundamental one, since technology constitutes the condition for our experience of time. But to what extent is this understanding of technology as recorded experience - that is, as memory - helpful in my investigation of software-based technologies and of software? If software is exteriorized memory, what does it record? Or – to rephrase the question – what would it mean to analyse software as a 'temporal object' (in Husserl's terms), or as 'a technical object' (in Stiegler's terms)?

To continue exploring the potentialities of the conceptual framework of originary technicity, and particularly of the notion of technical object, for my conceptualization of software let me now turn to one of Stiegler's key sources – that

[14] In fact, Stiegler introduces a technological dimension (which he calls 'technicity') into Husserlian 'primary retention'. In Hansen's words, for Stiegler 'tertiary memories – meaning, basically, all experience that has been recorded and is reproducible – represent our means of inheriting the past, the prosthetic already-there, and, for this reason, actually condition the other two forms of memory. Stiegler emphasizes the technical specificity of tertiary memory, for it is only once consciousness has the capacity to experience the exact same recorded experience more than once that we can appreciate how secondary retention (the memory of the first or earlier experience(s)) influences a subsequent primary retention' (Hansen 2003: non-pag.).

[15] Stiegler's complex argument, which mobilizes and transforms Husserl's theory of time-consciousness, is an extension of Derrida's own deconstruction of Husserl's distinction between primary retention and recollection (or secondary retention), and therefore between perception and imagination, and between presence and absence. Such deconstruction is the subject of Derrida's early studies on Husserl (Derrida 1973, 1978).

is, André Leroi-Gourhan's paleontological theory.[16] As I have mentioned above, Stiegler's idea of the technical object draws on Leroi-Gourhan's identification of the vertical posture, the use of the hand for purposes other than locomotion and the presence of tools and language as characteristics of the human species. While examining the early stages of the evolution of the human species, Leroi-Gourhan depicts locomotion as the determining factor of biological evolution (Leroi-Gourhan 1993: 25). For him freedom of the hand during locomotion implies the beginning of technical activity, in the same way that manual expertise frees the mouth from procuring nourishment and makes it available for speech. However simplified this synthesis, we can say that Leroi-Gourhan views evolution as a series of liberations taking place from the Paleozoic to the Quaternary eras, and from fish to human. Even more importantly, he asserts 'not only that language is a characteristic of humans as are tools, but also that both are the expression of the same intrinsically human property' (113). Language and tools evolve together, for they are 'neurologically linked and cannot be dissociated within the social structures of humankind' (114). In this context, Leroi-Gourhan proposes what is generally considered to be his fundamental contribution to anthropology and archaeology, that is the concept of operating sequence (*chaine operatoire*). He writes:

> Techniques involve both gestures and tools, sequentially organized by means of a 'syntax' that imparts both fixity and flexibility to the series of operations involved. This operating syntax is suggested by the memory and comes into being as a product of the brain and the physical environment. If we pursue the parallel with language, we find a similar process taking place.
>
> (Leroi-Gourhan 1993: 114)

The above formulation makes clear that the 'operating sequence' is a kind of sequential organization that underlies both language and technology. Therefore, for him, language and technology are the two sides of the same process ('the same intrinsically human property'), and evolve together. What appears problematic in

[16] Leroi-Gourhan's work left a powerful imprint on anthropology and paleontology in France. His 1964 book *Le Geste et la Parole* presents technology as a privileged point of access to the understanding of human evolution – in fact, as the pivot of a unified theory of human evolution. For Leroi-Gourhan, anthropology must be founded on technology, understood as the study of human material culture.

Leroi-Gourhan's theory – as Stiegler notices - is his explanation of the way in which the operating sequence comes into being:- that is, of how the 'intrinsically human property' that presides over technology and language originated at a certain point in time.

Leroi-Gourhan maintains that, with the emergence of *Homo sapiens*, the human species breaks up into ethnic groups, and a social apparatus based on cultural values appears. For Leroi-Gourhan, ethnicity is based on the concept of program. A program is realized in animals at the level of instinct, while in an ethnic group (or what could be more generally defined as a culture) it is expressed in the form of social values communicated through language. He puts it as follows:

> The whole of our evolution has been oriented toward placing outside ourselves what in the rest of the animal world is achieved *inside* by species adaptation. The most striking material fact is certainly the 'freeing' of tools, but the fundamental fact is really the freeing of the word and our unique ability to transfer our memory to a social organism other than ourselves.
>
> (Leroi-Gourhan 1993: 235)

Thus, Leroi-Gourhan establishes an equivalence between culture and program: both are to be understood as processes of exteriorization – that is, as the 'placing outside ourselves', and consequent socialization, of what in the rest of the animal world remains at the level of instinct. Importantly, this process of exteriorization, both of tools and of memory, enables Leroi-Gourhan to integrate contemporary technology into a unitary process of biocultural evolution. Today 'both tool and gesture are … embodied in the machine, operational memory in automatic devices, and programming itself in electronic equipment' (238). In the last stage of evolution, '*the hand is used to set off a programmed process* in automatic machines that not only exteriorize tools, gestures, and mobility but whose effect also spills over into memory and mechanical behaviour' (242). Leroi-Gourhan configures machines as the next step of human evolution. Today – he states – 'the main thrust of evolution is massively oriented toward tools' (251). For him, evolution has now entered a new stage, that of the 'exteriorization of the brain' (Leroi-Gourhan 1993: 252). He

envisions a possible future in which human beings become simply outdated, and the future of the species *as species* resides in intelligent machines.

It is worth emphasizing here that for Leroi-Gourhan not only does the exteriorization of memory explain culture - 'like tools, human memory is the product of exteriorization, and it is stored within the ethnic group' (258) - but that he also proposes a distinction between animal, human and mechanical memory that accounts for the evolution of the human species from the first kind of memory to the last. He writes:

> Animal memory is formed through experience within narrow genetic channels prespecialized by the species, human memory is constituted through experience based on language, and mechanical memory is constituted through experience within the channel of a preexisting program and of a code based on human language and fed into the machine by a human being.
>
> (Leroi-Gourhan 1993: 258)

Ultimately, Leroi-Gourhan is able to integrate the transmission of exteriorized (or collective) memory - that is, of culture - into a historical progression through the stages of oral transmission, written transmission and, finally, 'electronic transmission'. For him, computers belong to the latest phase, as the furthest example of the exteriorization of memory.

Up to this point, Leroi-Gourhan's theory proves rather interesting for my examination of software-based technologies, because it explicitly addresses the development of what I have called 'software-based technologies' – and what he calls 'intelligent machines' - from an anthropological point of view, and because it strives to position computers within the process of biocultural evolution. He identifies in the principle of the Jacquard loom the model for early automatic machines based on punched cards, and ultimately for computers. Interestingly, he depicts computers as gradually evolving from the model of the book. Computers are nothing but books that have become progressively autonomous from their human reader. It is worth quoting him at length on this point:

> Books in their 'raw' state are comparable to hand tools: however sophisticated their presentation, the reader's full technical participation is still required. A simple card index already corresponds to a hand-operated machine: some of the operations have been transformed and are now contained in potential form in the index cards, which are the only things the reader needs to activate. Punched index cards represent yet another stage, comparable to that of early automatic machines. ... The data are converted by means of a binary code (positive = no perforation, negative = open perforation), and a sorting device separates the cards according to a set of questions, releasing only those that produce an affirmative response. The principle is that of the Jacquard loom, and it is curious to note that documentary material waited for more than a century to follow in the footsteps of weaving. ... A punched-card index is a memory-collecting machine. It works like a brain memory of unlimited capacity that is endowed with the ability – not present in the human brain – of correlating every recollection with all others. ... The electronic brain, although it employs different and more subtle processes, operates on the same principles.
>
> (Leroi-Gourhan 1993: 264)

As the above passage shows, for Leroi-Gourhan the process that leads to the autonomization of the book takes the form of an exteriorization of ordering operations, by which the book gradually becomes an automated device capable of sorting out documentary material according to the same principles that inform loom weaving. Computers represent the latest stage of this evolution.

The role of the loom in the history of the computer is well known (see, for instance, Morrison and Morrison 1961). To give but one example, Sadie Plant writes that '[t]he loom is the vanguard site of software development' (Plant 1995: 46; see also Plant 1998). Weaving is a complex process that involves integrating several threads into one cloth; thus, Fernand Braudel describes the loom as 'the most complex human engine of them all' (Braudel 1973: 247). Jacquard's automated loom was based on the principle of punched cards, where the threads selected by each card

were the ones that passed through its holes. This principle was not new (it had been used since early 18th century), but Jacquard strung the cards together in sequences, each of which constituted an ordered set of weaving operations – that is, in Leroi-Gourhan's words, a program. For this reason, when in the 1840s Charles Babbage started working on his Analytical Engine (which is generally considered the prototype of the computer), he modelled it on Jacquard's strings of punched cards. His own contribution was to introduce 'the possibility of bringing any particular card or set of cards onto use *any number of times successively in the solution of one problem*' (Morrison and Morrison 1961: 264). Thus, Babbage considered his machine as characterized by memory and foresight - that is, by the possibility of referring to past operations in order to act in the future (Morrison and Morrison 1961). However limited, this account of the relationship between the early prototypes of the computer and the principles of the loom shows the presence of an explicit connection between memory and time even in the early days of computing.

I want to suggest that Leroi-Gourhan's fascinating theory should be credited for stimulating the reconceptualization of software-based technologies within a general frame of reference that escapes the concept of instrumentality and that regards technology as constitutive of the human. In fact, I will come back to the important connection established by Leroi-Gourhan between language and technology in the second section of this chapter, in which I will analyse Derrida's reworking of Leroi-Gourhan's thought and evaluate the relevance of such reworking for the study of software. Yet for now it is important to notice that Leroi-Gourhan's theory does not seem able to answer the very question it opens up – namely, the question of the emergence of the 'operational sequence' (and therefore of language, technology, and ultimately of humanity) as a result of the interaction between the brain and the physical environment. Therefore, as Stiegler shows in the first volume of *Technics and Time* (1998a), it is precisely as a general frame of reference – that is, as a theory of originary technicity - that Leroi-Gourhan's thought proves less satisfactory. Let me now turn to Stiegler's critique of Leroi-Gourhan in order to continue apprising the relevance of the framework of originary technicity for my investigation of software.

Although Stiegler's understanding of technology as 'exteriorization' parallels Leroi-Gourhan's paleontological theory, in *Technics and Time* Stiegler clearly demonstrates that Leroi-Gourhan's account of the co-evolution of the human and the technological falls prey to the same difficulty that, in the eighteenth century, Jean-Jacques Rousseau confronted in 'The Origin of Languages' - namely, the classical question of 'the origin of man' (Stiegler 1998a: 117-141). To simplify Stiegler's argument, Rousseau developed a providentialist explanation of the origins of humanity and of society, since he could not explain the passage to the human except through the intervention of God (Stiegler 1998a: 115; Beardsworth 1995: 6). According to Stiegler, in Leroi-Gourhan's theory technology is similarly the *écart* (or the rupture) that starts the process of hominization - that is, the process through which the human species escapes its own genetic program, thus opening up the horizon of humanity. But Leroi-Gourhan's theory falls short precisely of explaining this passage from the genetic to the non-genetic (and thus the passage to the human). Its contradictory treatment of the prehominid (the Australopithecus, and more accurately the Zijanthropicus) contributes to this impasse.

Put briefly, according to Stiegler Leroi-Gourhan grants the prehominid the possibility of speech and of tool use, but he refuses it the status of the human. For Leroi-Gourhan the prehominid is non-human because it has no faculty of 'anticipation' (namely, of thinking its own death), which in turn Leroi-Gourhan assumes to be the basis of what he calls 'the symbolic' - that is, culture. Stiegler unmasks here Leroi-Gourhan's attempt to define culture as something altogether different from technology - i.e. as a qualitative leap from the capability of using tools to the capability of using symbols. This is obviously in flagrant contradiction with Leroi-Gourhan's main argument that humanity is nothing but a process of exteriorization accomplished through technology (Stiegler 1998a: 152).[17]

[17] Stiegler's reinterpretation of Leroi-Gourhan is an example of deconstructive reading, and is profoundly indebted to Derrida's work. However, Richard Beardsworth (1995) points out that Stiegler's rereading of Leroi-Gourhan differs from Derrida's in their treatment of hominization. For Derrida hominization - that is, the origin of the human - is unattainable and cannot be narrated; such a narrative would only be fictional. Stiegler in turn maintains that it is possible to recount the origin of the human in terms of the mutual constitutivity of the human and the technical as 'organized inorganic matter'. I will return to this concept later on in the present chapter.

Stiegler proposes his own answer to the dilemma of the origin of technology and of the human. His solution is based on the concept of the 'technical object' as 'organized inorganic matter' (*matière inorganique organisée*) – a further clarification of the idea of technology as the support for consciousness. Richard Beardsworth (1995) clarifies how Stiegler understands the material aspects of technology as constitutive of the human. He explains:

> Organized inorganic matter is matter which transforms itself in time as technical object. Whilst in time, its transformations, however, are the condition of the human temporalization of time. In this sense, matter is constitutive of temporality. And this, in an explicit historical sense: each 'time' matter undergoes radical evolution, the temporalization of time changes.
>
> (Beardsworth 1995: non-pag.)

I want to emphasize here that Stiegler's concept of the technical object holds two orders of consequences: it allows us to think the transformation of technology through time, while at the same time exposing the crucial role that technology plays in the constitution of the human experience of time. According to Stiegler, material objects from the stone instrument to the portable computer change our way of perceiving time, and therefore affect the emergence of our identity and our understanding of what it means to be human. The human can thus only be thought in relation to material technologies that change over time, changing in turn what it means to be human. Let me now analyse some of the implications of this part of Stiegler's argument for my investigation of software.

First of all, for Stiegler the co-implication with materiality distinguishes human beings from other forms of life. In other words, human beings have a unique relationship to their environment, and this relationship is mediated by technology. Technical objects represent the interface between the human and its environment. Moreover, technical objects, being at the same time organized and inorganic, are neither living matter nor inert matter. They change through a specific kind of evolution. In Stiegler's words:

> The zootechnological relation of man to matter is a particular case of the relation of the living to the environment, that is, a relation of man to his environment which passes through organized inert matter, the technical object. The singularity of this relation is that the inert, although organized matter which is the technical object evolves itself in its organization: it is no longer simply inert matter, but it is not living matter either. It is an organized inorganic matter which is transformed in time, just as living matter is transformed in interaction with the environment. Moreover, it becomes the interface through which the living matter which is man enters into relation with the environment.
>
> (Stiegler 1998a: 63)

Stiegler's articulation of matter as 'inorganic organized matter' allows for a history of human culture as – in Beardsworth's terms - 'the history of the differentiation of the originary complex human-technical' (Beardsworth 1995: non-pag.). Beardsworth's formulation is significant because it shows exactly how Stiegler resolves Leroi-Gourhan's impasse on the origin of technology. For Stiegler the mutual constitutivity of technology and the human makes it impossible to decide which is the origin of the other. Nevertheless, we can tell the history of how this reciprocal ongoing constitution takes place over time. We can investigate how human beings and technology co-evolve without having to decide which one to prioritize. Stiegler names this process within which the human and the technical mutually constitute each other 'the originary complex who-what' (Stiegler 1998a: 142). Beardsworth (1995) clarifies this point further:

> The who of the human species is nothing less than the who-what of the reflective relation between cortex and tool. This who-what is both a differentiation within life (starting with the stone implement) and itself a constant process of differentiation (the history of technics). It is in these terms that the passage from the genetic to the non-genetic is to be understood.
>
> (Beardsworth 1995: non-pag.)

To summarize, human beings, technology and culture are part of the same process of exteriorization, which, as we have seen earlier, Stiegler names 'ephiphylogenesis'. For him, we have emerges as human beings as a 'result' of three kinds of memory: our genetic memory, the 'individual' memory - or the memory of our central nervous system (which is responsible for our remembering of experiences, and which Stiegler names 'epigenetic'), and the techno-logical ('epiphylogenetic') memory, which preserves the experiences of past generations in the tools and language we inherit from the past, and therefore is an 'externalized', shared memory. In Stiegler's words:

> Epiphylogenesis, a recapitulating, dynamic and morphonegentic (phylogenesis) accumulation of individual experience (epi) designates the appearance of a new relation between the organism and its environment, which is also a new state of matter. If the individual is organic organized matter (*une matière organique et organisée*), then its relation to its environment (to matter in general, organic or inorganic), when it is a question of a who, is mediated by the organized but inorganic matter of the organon, the tool with its instructive role (its role as instrument), the what. It is in this sense that the what invents the who just as much as it is invented by it.
>
> (Stiegler 1998a: 185)

This passage clarifies that only with epiphylogenesis the human being reaches a new relationship with its environment – a relationship mediated through technology ('the technical object'). Technology carries with it memories of the past - not only of the individual past, but of the past generations. In this sense, it can be said that the 'who' (the human being) invents technology, but at the same time it is invented ('instructed') by it, by the memory of past experiences that technology carries. This is the sense of the mutual co-constitution of the 'who' and the 'what', of technology and the human. Stiegler's answer to Leroi-Gourhan anthropological impasse on technology is to emphasize that there is actually no way to distinguish between the material aspects of technology and the temporality it carries with it – and therefore there is no way to separate technology from culture, or technology from the human.

But what would it mean to investigate software as 'organized inorganic matter'? What would the consequences of thinking software within the framework of originary technicity be? In what way is software a 'what' that constitutes the 'who' that interacts with it? In what way is one constituted by software as much as one produces and uses it? To explore these issues further let me now examine how Stiegler develops his theory in relation to new technologies' in the third volume of *Technics and Time.* For him, new technologies constitute a major break in the history of technology. He explains:

> [The] independence of mnenmotechnics from the technical system of production no longer exists today: in becoming planetary, the technical system is now also, and even foremost, a global mnemotechnical system. In a sense, a fusion between the technical system, the mnemotechnical system and globalization has occurred. ... *The global technical system has basically become a mnemotechnical system for the industrial production of tertiary retentions*, and thus for *criteria* of retentional selection, of the flux of consciousness inscribed into processes of adoption.
>
> (Stiegler 2003a: non-pag.)

This complex passage is based on Stiegler's distinction between 'technics' and 'mnemotechnics'. For him, while all kinds of technology always transmit memories, some have been produced expressly with a view to transmitting memories. Stiegler gives the name of 'mnemotechnics' to technologies specifically devoted to the transmission of memory (for instance, writing, photography, phonography and cinematography). '[T]echnics is always a memory aid', he maintains, '– this is what we mean by epiphylogenesis. But not every technics is a mnemo-technics. The first mnemotechnical systems appear after the Neolithic period. They form what will later become the kind of writing we are still using today' (Stiegler 2003a: non-pag.).

In Stiegler's view, technical systems precede mnemotechnical systems. When he investigates the historical transformation of technology, he focuses on technical systems - that is, mainly technological systems of production. In the first volume of *Technics and Time* he deploys Bertrand Gille's concept of 'technical system' as a

moment of stability in time, or a point of equilibrium in the process of technical change that characterizes history. This point of equilibrium is expressed in a particular technology. In other words, every civilisation constitutes itself around a technical system, which is in turn organized around a dominant technology. Every technical system has in itself a potential for change, and actually undergoes evolutionary transformations and periods of crisis. During a crisis, a technical system evolves at great speed, causing 'dis-adjustments' with the other social systems – such as economy, politics, education, and so forth, and it can only return to (relative) stability when these other systems have 'adopted' the new technical system (Stiegler 2003a: non-pag.).

I have shown earlier on in this chapter how for Stiegler the contemporary globalised industrial technical system (whose beginnings took root in England at the end of the eighteenth century) has entered an epoch of permanent innovation, becoming 'fundamentally unstable'. Such globalisation of the industrial technical system has been made possible, to a great extent, by information and communication technologies, which facilitate, for instance, the automation and control of remote production and distribution, the international circulation of capital and the opening up of intercontinental markets. Stiegler refers to this globalised system as 'a single planetary set-up' [*dispositif*] (Stiegler 2003a: non-pag.). But why does this set-up constitute a break in the history of technology?

Such a break is due precisely to the newly acquired importance of information and communication technologies. Until recently, mnemotechnics had always evolved slower than the technical systems of material transformation. While the latter underwent substantial changes from the age of ancient Greeks to the Industrial Revolution, alphabetic writing remained more or less stable. This independence has now ceased to exist, since communication and information industries have become the centre of the technical system of production and – more generally – the decisive element of the global technological system. This has in turn led to a change in our perception of space and time. For instance, the distances and the delay in the circulation of messages tend to be nullified by global networks, and 'night and day become interchangeable through artificial electric light and computer screens'

(Stiegler 2003a: non-pag.). Our mechanisms of orientation are therefore profoundly disturbed.

The question of software as inorganic organized matter becomes thus the question of the place of software in the globalised mnemotechnical system. Such a question needs to be reformulated as follows: what kind of mnemotechnics is software? And – even more importantly – is Stiegler's distinction between technics and mnemotechnics meaningful for an investigation of software in the first place? To discuss this extremely important point, let me look again at Clark's concept of originary technicity which I have examined earlier on in this chapter (Clark 2000). While commenting upon the novel *The Turing Option*, Clark describes technology as 'systems of signs'; thus, for him originary technicity seems to have an intrinsic relation to what Stiegler calls mnemotechnics. Conversely, Stiegler's distinction between the two is based on the following argument: every technics (for instance, pottery) carries the memory of a past experience; but only mnemotechnics (for instance, writing) are conceived with the primary *purpose* of carrying the memory of a past experience. In Stiegler's argument, the emphasis is on the aim, or end, of different technologies: some technologies are conceived just for recording, others are not.

At this point I want to risk the following proposition: software transgresses Stiegler's distinction between technics and mnemotechnics. Although this thesis needs to be proved, it is important to position it first of all as a problem. Let me explain this point briefly. If one relies on the widely accepted definition of software that constitutes the foundation of Software Engineering, for instance, one finds that 'software' is the totality of all computer programs as well as all the written texts related to computer programs (Humphrey, 1989a; Sommerville, 1995). To give but one example, Ian Sommerville writes that 'software engineers model parts of the real world in software. These models are large, abstract and complex so they must be made visible in documents such as system designs, user manuals, and so on. Producing these documents is as much part of the software engineering process as

programming' (Sommerville, 1995: 4).[18] According to this definition, software can be thought of as a totality of 'documents' or 'texts' written in natural and formal languages at every stage of software development. Thus, software can be considered – in Stiegler's terms – as mnemotechnics. On the other hand, it cannot be said that the main purpose of software is recording in the same way that it is for writing or cinema. It could be argued that the main purpose of software is to make things happen in the world (for instance, to change the polarities of the electronic circuits within a computer on which software is executed). This is why software might be the point where Stiegler's distinction between technics and mnemotechnics is suspended.

A correlated question arises at this point: by introducing the distinction between technics and mnemotechnics is Stiegler involuntarily reintroducing the separation between the technical and the symbolic that he deconstructs in Leroi-Gourhan? Certainly, in the third volume of *Technics and Time*, Stiegler speaks of a convergence between technics and mnemotechnics, but in a much more general sense - that is, as a convergence of technologies of production with information and communication technologies. For him, undoubtedly, information and communication technologies fall under the rubric of mnemotechnics – or technology that has recording as its primary aim. I want to argue that ultimately, in order to distinguish between technics and mnemotechnics, Stiegler resorts to the concept of the aim (or the end) of technology, therefore seemingly falling back into an instrumental conception of technology – which obviously contradicts his understanding of technology as originary.

But, if software puts in crisis the distinction between technics and mnemotechnics, how is one supposed to think software within the framework of originary technicity? In what way is the relationship between the technical and the symbolic articulated in software? As we have seen, Clark states that the thinkers of originary technicity situate the question of technology 'in the subtle intimacy' of the relation

[18] As I highlighted in the Introduction, software has never been univocally defined by any disciplinary field. However, the definition of software provided by Software Engineering is a very general one – as it can be expected from a discipline that was established in the late 1960s with the purpose of helping programmers designing software cost-effectively regardless of the specific applications and programming languages they were working with.

between technology and language (Clark 2000: 240). Moreover, according to Clark, Jacques Derrida is one of the most important thinkers of originary technicity precisely because he 'takes on the radical consequences of conceiving technical objects (including systems of signs) as having a mode of being that resists being totally understood in terms of some posited function or purpose for human being' (Clark 2000: 240). By his refusal to explain either technology or language in instrumental, functionalist terms Derrida resists the widespread denigration of the 'merely' technical in Western thought. In order to continue exploring the possibility of conceptualizing software within the framework of originary technicity, let me now leave Stiegler's thought for a little while and turn to the examination of Derrida's understanding of technology.

Derrida makes references to technology and to the importance of technicity for the definition of the human throughout his whole work. Importantly, his conception of technology as something that cannot be understood within the conceptual framework of instrumentality is inseparable from his understanding of writing. Actually Derrida traces the devaluation of instrumentality back to the famous devaluation of writing that I have examined earlier on in Plato's *Phaedrus* (Derrida 1981). For Derrida, as for Stiegler, the devaluation of instrumentality cannot be separated from the devaluation of writing.

Derrida's reflection on writing is crucial to the whole of his theory, and lies at the core of his criticism of Western metaphysics. Derrida's goal is not a reversal of priorities - namely, the prioritizing of writing over speech - but a critique of the whole of Western metaphysics that he understands as 'logocentric'. As Gayatri Spivak points out in her introduction to *Of Grammatology*, the term 'writing' is used by Derrida to name a whole strategy of investigation, not merely 'writing in the narrow sense' as a kind of notation on a material support (Derrida 1976: lxix). Thus, Derrida writes *Of Grammatology* not to pursue a mere valorisation of writing over speech, but to present the repression of writing 'in the narrow sense' as a symptom of logocentrism that forbids us to recognize that everything is pervaded by the

structure of 'writing in general' - that is, an eternal escaping of the 'thing itself'.[19] Derrida argues that speech too is structured like writing. There is no structural distinction between writing and speech – except that, in the history of metaphysics, writing has been repressed and read as a surrogate of speech.

In the chapter 'The end of the book and the beginning of writing' of *Of Grammatology* Derrida maintains that today writing can no longer be thought as 'a particular, derivative, auxiliary form of language in general', or as 'an exterior surface, the insubstantial double of a major signifier, *the signifier of the signifier*' (Derrida 1976: 7). Making writing instrumental is a move of Western metaphysics, and it is paired with the notion of speech as fully present. From this perspective, writing is seen as an interpretation of original speech, as technology in the service of language. However, Derrida suggests that language could only be a 'mode' or an aspect of writing.

I will return to Derrida's complex argument in Chapter Two in greater detail. For now, suffice it to say that Derrida's questioning of logocentrism is inseparable from his questioning of the instrumental conception of technology. In *Mémoires: for Paul de Man* he states that '[t]here is no deconstruction which does not ... begin by calling again into question the dissociation between thought and technology, especially when it has a hierarchical vocation, however secret, subtle, sublime or denied it may be' (Derrida 1986: 108). Thus, once again, Derrida makes it explicit that the dissociation between thought and technology is – as is every other binary opposition - hierarchical, since it implies the devaluation of one of the two terms of the binary – in this case, technology. For this reason Clark (2000) suggests that 'originary technicity' can be considered another name for Derrida's 'writing in the general sense'. As Derrida states in *Of Grammatology*:

> Writing is not an auxiliary in the service of science – and possibly its object – but first, as Husserl in particular points out in *The Origin of Geometry*, the condition of the possibility of ideal objects and therefore

[19] On the other hand, in the section of *Of Grammatology* about Lévi-Strauss, Derrida (1976) suggests that no definite distinction between writing in the 'narrow' and the 'general' sense can be traced, for one slips into the other.

> of scientific objectivity. Before being its object, writing is the condition of the *episteme.*
>
> (Derrida 1976: 27)

This passage is crucial for clarifying the relationship that Derrida establishes between writing and thought, and ultimately for his understanding of technology as constitutive of the human. As Clark explains, for Derrida 'writing enregisters the past in a way that produces a new relation to the present and the future, which may now be conceived within the horizon of an historical temporality, and as an element of ideality' (Clark 2000: 241).[20] Thus, the written mark gives us the possibility of keeping trace of the past and enables us to acquire a sense of time. Clearly Derrida views writing – understood here as technology, or the technological capacity of registering the past - as a constitutive condition of thought. Consequently, technology cannot be understood through the opposition between *technê* and *epistêmê*, because it precedes and enables such an opposition. But what would all this mean for an investigation of software? To be more specific, in what way would the reformulation of 'originary technicity' in terms of Derrida's 'writing in general' advance my investigation of software? This reformulation of the problem amounts on the one hand – as we have already seen - to asking what the significance of the study of software for an understanding of the relationship between technology and the human is. On the other hand, it opens up the methodological question of whether and in what way software should be approached as a historically specific technology. I want to argue that, in order to start addressing both of these questions, it is useful to examine Derrida's rereading of Leroi-Gourhan's work in *Of Grammatology.*

For Derrida, Leroi-Gourhan has shown in *Le geste et la parole* that the historical perspective that associates humanity with the emergence of writing (and therefore excludes peoples 'without writing' from history) is profoundly ethnocentric. In fact, it shortsightedly denies the characteristic of humanity to peoples that do not actually lack 'writing', but only 'a certain type of writing' (Derrida 1976: 83) - that is, alphabetic writing. To explain this point Derrida draws on Leroi-Gourhan's concept of 'linearization'. For Leroi-Gourhan the emergence of alphabetic writing must be

[20] Again, I will return to Derrida's rereading of Husserl's phenomenology in Chapter Two.

understood as a process of linearization (Leroi-Gourhan 1993: 190). In his analysis of the emergence of graphism, Leroi-Gourhan emphasises what he considers to be the underestimated link between figurative art and writing. '[I]n its origins', he states, 'figurative art was directly linked with language and was much closer to writing (in the broadest sense) than to what we understand by a work of art' (190). Given the difficulty of separating primitive figurative art from language, he proposes the name 'picto-ideography' for this general figurative mindframe. Yet he is very clear that such a mindframe does not correspond to writing 'in its infancy' (195). Such an interpretation would amount to applying to the study of graphism a mentality influenced by four thousand years of alphabetic writing – something that linguists have often done, for instance, when studying pictograms. But 'picto-ideography' signals an originary independence of graphism from the mental attitude that constitutes the basis of what Leroi-Gourhan calls 'linearization'.

To understand the concept of linearization better, one must start from Leroi-Gourhan's concept of language as a 'world of symbols' that 'parallels the real world and provides us with our means of coming to grips with reality' (195). For Leroi-Gourhan graphism is not dependent on spoken language, although the two belong to the same realm. Leroi-Gourhan views the emergence of alphabetic writing as associated with the technoeconomic development of the Mediterranean and European group of civilizations. At a certain point in time during this process writing became subordinated to spoken language. Before that – Leroi-Gourhan states - the hand had its own language, which was sight-related, while the face possessed another one, which was related to hearing. He explains:

> At the linear graphism stage that characterizes writing, the relationship between the two fields undergoes yet another development. Written language, phoneticized and linear in space, becomes completely subordinated to spoken language, which is phonetic and linear in time. The dualism between graphic and verbal disappears, and the whole of human linguistic apparatus becomes a single instrument for expressing and preserving thought – which itself is channelled increasingly toward reasoning.
>
> (Leroi-Gourhan 1993: 210)

By becoming a means for the phonetic recording of speech, writing becomes a technology. It is actually placed at the level of the tool, or of 'technology' in its instrumental sense. As a tool, its efficiency becomes proportional to what Leroi-Gourhan views as a 'constriction' of its figurative force, pursued precisely through an increasing linearization of symbols. Leroi-Gourhan calls this process 'the adoption of a regimented form of writing' that opens the way 'to the unrestrained development of a technical utilitarianism' (212).

Expanding on Leroi-Gourhan's view of phonetic writing as 'rooted in a past of nonlinear writing', and on the concept of the linearization of writing as the victory of 'the irreversible temporality of sound', Derrida relates the emergence of phonetic writing to a linear understanding of time and history (Derrida 1976: 85). For him linearization is nothing but the constitution of the 'line' as a norm, a model – and yet, one must keep in mind that the line is *just* a model, however privileged. The linear conception of writing implies a linear conception of time - that is, a conception of time as homogeneous and involved in a continuous movement, be it straight or circular. Derrida draws on Heidegger's argument that this conception of time characterizes all ontology from Aristotle to Hegel - that is, all Western thought. Therefore, and this is the main point of Derrida's thesis, 'the meditation upon writing and the deconstruction of the history of philosophy become inseparable' (86).

However simplified, this reconstruction of Derrida's argument demonstrates how, in his rereading of Leroi-Gourhan's theory, Derrida understands the relationship of the human with writing and with technology as constitutive of the human rather than instrumental. Writing has become what it is through a process of linearization - that is, by conforming to the model of the line – and in doing so it has become instrumental to speech. Since the model of the line also characterizes the idea of time in Western thought, questioning the idea of language as linear implies questioning the role of the line as a model, and thus the concept of time as modelled on the line. It also implies questioning the foundations of Western thought (by means of a strategy of investigation that, as we have seen, Derrida names 'writing in general', or 'writing in the broader sense'). At this point it becomes clear why, if we

follow Derrida's reworking of the concept of originary technicity, a new understanding of technology (as intimately related to language and writing) entails a rethinking of Western philosophy – ambitious as this task may be.

It is worth noting here that in *Of Grammatology* Derrida expressly highlights how the reconceptualization of the Western tradition of thought is particularly urgent today. Such a rethinking is what Derrida famously calls 'the end of the book', or the end of linear writing. According to Derrida, we are suspended today between two eras of writing – and this is why we can also reread our past differently. Actually, the 'uneasiness' of philosophy in the past century is due to an increasing destabilization of the model of the line. He states that what is thought today cannot be written in a book - that is, it cannot be thought through with a linear model - any more than contemporary mathematics can be taught with an abacus (87). This inadequacy does not only apply to the current moment in time, but it comes to the fore today more clearly than ever. Derrida writes:

> The history of writing is erected [by Leroi-Gourhan] on the base of the history of the *grammé* as an adventure of relationships between the face and the hand. Here, by a precaution whose schema we must constantly repeat, let us specify that the history of writing is not explained by what we believe we know of the face and the hand, of the glance, of the spoken word, and of the gesture. We must, on the contrary, disturb this familiar knowledge, and awaken a meaning of hand and face in terms of that history.
>
> (Derrida 1976: 84)

I will return to Derrida's emphasis on the urgency of re-reading Western philosophy in relation to new technologies in Chapter Two. For now suffice it to say that, as the above passage makes quite obvious, for Derrida what is most relevant in Leroi-Gourhan's history of writing is that it problematizes our conception of the human ('what we believe we know of the face and the hand'). Yet the focus of Derrida's work is not the concrete analysis of historical systems of writing, since, as we have seen, he differentiates 'writing in general' from any such system. With regard to my investigation of software, then, Derrida's understanding of what Clark calls

'originary technicity' has two important implications. On the one hand, it confirms the fundamental relationship between technology and the human, and it supports the need for a radical questioning of both concepts - and ultimately of Western thought. On the other hand, Derrida leaves open the question of how to investigate a historically specific technology (for instance, software) without losing its significance for a radical rethinking of the relationship between technology and the human.

What I want to suggest now is that, in *Technics and Time*, Stiegler rereads Derrida's thought on originary technicity in a way that allows for precisely such an investigation. To illustrate this point further, I want to start from Hansen's assertion that 'Stiegler's conception of the originary prostheticity of the human, of its co-emergence with tool use, performs a certain displacement of deconstruction' (Hansen 2003: non-pag.). Such displacement has the effect of specifying, even beyond Derrida's own conception, the constitutive technicity of 'writing in general' (Hansen 2003: non-pag.). In fact, according to Hansen, Stiegler reinterprets Derrida's thought so that writing in general cannot be separated from its appearance in a concrete technical inscription system. Let me now explain this point a little further, in order to establish to what extent and in what way software can be investigated as a historically specific technology.

As Hansen points out, for Stiegler transcendence 'is constituted through technics as the support for the inscription of memory' (Hansen 2003: non-pag.). Beardsworth (1995) regards this point as Stiegler's 'break' with Heidegger. He acknowledges that 'according to Stiegler ... it is technics which, as the support of the inscription of memory, is constitutive of transcendence' (Beardsworth 1995: non-pag.). These comments by Hansen and Beardsworth need to be carefully examined. Not only does Stiegler establish a relationship between technology and the possibility of thought: he also articulates this relationship historically by arguing that *technê* and *epistêmê* – or, as I have shown earlier on in this chapter, 'technology' and 'knowledge' - are correlated through writing. In fact, for Stiegler 'philosophical' problems – that is, problems that historically arise with Thales and are associated with philosophy as non-technical knowledge - 'fundamentally proceed from the appearance of *a techno-logy*', and specifically a technique of writing (Stiegler

2003b: 154). The emergence of the technique of linear writing radically transforms the modes of cultural transmission from generation to generation, Stiegler argues. In fact, from the point of view of Greek pre-Socratic thought, which does not presume the immortality of the soul, the dead can nevertheless return as ghosts that transmit an inheritance, and such inheritance is deemed to come from a spirit (*esprit*) that crosses generations. This is the pre-Socratic image of cultural transmission. In contrast, the appearance of linear writing allows for the transmission of culture 'as a unified spirit, precisely through the unification of language enabled by literalization' (154). Drawing on Leroi-Gourhan's and Derrida's thought, Stiegler insists that the emergence of the model of the line has changed both the transmission of culture *and* the modes of thought. He writes:

> It is this mnemotechnics that makes possible the writing of laws, the founding of cities, the construction of geometric reasoning (Thales embodies the origin of geometry), the practice of philosophy. It involves a massive transformation of the social group that raises a thousand questions. It overturns, for example, the relation to tradition, to spirits, and, more precisely, the articulation between the city and religion, between the profane and the sacred, the place of the clans inside the city-states or territories [*demes*], and so on.
>
> (Stiegler 2003b: 154)

According to Stiegler, the sophists themselves are a by-product of this process. The years between the seventh and the fifth century BCE are witness to the arrival of the *grammatists*, the masters of letters, and later on of the sophists, who 'go on systematically to develop a technique of language that quickly acquires a critical dimension, in so far as this technique of developed language will in turn engender a moral crisis' (155). Thus, sophistry is not an oral technique; rather, it presupposes writing.[21] Accordingly, Plato criticizes the sophists because they manage to speak well, 'but they learn everything by heart, by means of this techno-logical "hypomnesis" that is logography, the preliminary writing out of speeches. It is

[21] Stiegler's assertion mirrors Derrida's argument that we need to have a sense of writing in order to have a sense of orality. I will develop this point further in Chapter Two.

because writing exists that the sophists can learn the apparently "oral" technique of language that is rhetorical construction' (155). In the *Ion* Plato even makes a connection between poets and sophists, claiming that they work along the same lines of falsehood: '[s]ophists, poets, are only liars, *that is to say*, technicians' (155). This powerful image of the technician as a liar constitutes the summation of Plato's devaluation of technology and writing.

To summarize, Stiegler points out that, on the one hand, the question of technology, considered as the object of repression, 'is a question that emerges *with* and *by* its denunciation by Plato' (155). It arises 'above all as a *denial*, and in this sense therefore as *a kind of forgetting'* – and this is quite paradoxical, since in *Phaedrus* what Plato blames technology for is precisely its power of forgetting (155). On the other hand, it can be said that the question of technology appears well before Plato: as we have just seen, it arises in the context of the transformation of the Greek cities, associated with the development of navigation, money, and above all mnemotechnics, that is to say of technologies capable of transforming the conditions of social and political life and of thought. Ultimately, *technê* and *epistêmê* – that is, knowledge and technology - share a relationship with writing, the fundamental mnemotechnics. In turn, mnemotechnics - and technology in general - reveal a constitutive connection with temporality. Let me now go over Stiegler's argument one last time, in order to show how it provides a consistent foundation for the concrete study of the transformation of technology in time – and, ultimately, for the investigation of software as a historically specific technology.

As I have remarked earlier on, Stiegler's understanding of the transformation of technology in time is crucially related to his 'displacement' of deconstruction that also results in his break with Heidegger (Hansen 2003: non-pag.). Stiegler explains:

> Let's say, for example, that one night I write the sentence: 'it is dark'. I then reread this sentence twelve hours later and I say to myself: hang on, it's not dark, it's light. I have entered into the dialectic. What is to be done here? ... That which makes consciousness be self-consciousness (i.e. consciousness that is conscious of contradiction with itself) is the fact that consciousness is capable of externalising itself.
>
> (Stiegler 2003b: 163)

This passage is extremely important because it reformulates the concept of the technical constitution of consciousness that Clark explores in his analysis of *The Turing Option* (Clark 2000: 240), to which I have repeatedly referred throughout this chapter. Here what Stiegler - and Leroi-Gourhan before him - call 'exteriorization' (which constitutes the basis of self-consciousness) is clearly pursued *through writing*. One writes 'it is dark', and when one rereads the note twelve hours later it is light. This produces, as Stiegler himself further clarifies, 'a contradiction between times', namely the time of consciousness when one wrote this and the time of consciousness when one reads this. Yet, one has the same consciousness, which is therefore 'put in crisis' (Stiegler 2003b: 163), and this crisis in turn raises self-awareness. The act of inscription - that is, of exteriorization - ultimately constitutes interiority, which does not precede exteriority, and vice versa. As I have explained earlier, for Stiegler (again drawing on Leroi-Gourhan) the process of exteriorization constitutes the foundation of temporality, of language and of technical production, and requires a basic neurological 'competence' - that is, 'a level of suitable cortical and subcortical organization' (164).

Importantly, there exist three kinds of temporality: physical temporality (that is, the temporality of the universe, which expands entropically), biological temporality (that is, the temporality of the living, which fights against disorder to maintain its organization) and technical temporality. In particular, as we have already seen, technical temporality is constituted through the technical object, and it both 'extends and breaks with the living', since it allows the human species to transcend its genetic determination. This is fundamentally the sense of Stiegler's famous reinterpretation of the myth of Prometheus and Epimetheus (Stiegler 1998a, 2003b). This myth, narrated in Plato's *Protagoras*, shows that human beings are not predestined to be what they are.[22] On the contrary, they are 'prosthetic' - that is, endowed with artefacts and capable of altering them, and therefore also capable of altering their destiny. It is this capacity that for Stiegler distinguishes – and actually constitutes - the human experience of time.

[22] Plato, *Protagoras* 320d ff. (Plato 1989)

This is Stiegler's fundamental point of departure from Derrida's theory. Through this departure he lays the foundation for the concrete study of historically specific technologies as fundamental to the understanding of the constitutive relationship between technology and the human. To clarify this point, it is now worth examining Stiegler's interpretation of the myth of Prometheus and Epimetheus briefly. According to the myth, Zeus gives Prometheus the task of distributing qualities and powers to the living creatures, but Prometheus leaves it to his twin brother Epimetheus to act in his place. Epimetheus hands out all the qualities to the living and forgets to keep one for the human being. Human beings therefore appear here as characterized by a 'lack of quality' (Stiegler 2003b: 156). Stiegler comments that the human being is 'a being by default, a being marked by its own original flaw or lack, that is to say afflicted with an original handicap' (156). For this reason, Prometheus decides to steal technology - that is, fire - and gives it to human beings, in order to enable them to invent artefacts and to become capable of developing all qualities. With the gift of technology, a problem arises: mortals cannot agree on how to use artefacts, and consequently start fighting and destroying each other. In Stiegler's words, '[t]hey are put in charge of their own fate, but nothing tells them what this fate is, because the lack [*défaut*] of origin is also a lack of purpose or end' (156). Stiegler's reworking of the myth clearly demonstrates how for him technology raises the problem of decision, and how this encounter of the human with decision in turn constitutes time - or better, what Stiegler calls 'technical time'. Technical time emerges because human beings experience their capacity of making a difference in time through decision. Temporality is precisely this opening of the possibility of a decision, which is also the possibility of creating the unpredictable, the new.

It is for this very reason that the historical specificity of technology is central to Stiegler's thought. The human capability of deciding 'what to become' *constitutes* temporality. Moreover, human prostheticity - that is, the fact that human beings, to survive, require non-living organs such as houses, clothes, sharpened flints, and all that Stiegler calls 'organized inorganic matter' - forms the basis for memory, or better, for technical memory. As I have shown earlier on, unlike genetic and individual memory, technical memory coincides with the process of exteriorization that 'enables the transmission of the individual experience of people from

generation to generation, something inconceivable in animality' (159). This inherited experience is what Stiegler calls 'the world' - that is, a world that is always already haunted by 'spirits' in the pre-Socratic sense, always already constructed by the memories of others.

As I have pointed out earlier on in this chapter, Stiegler departs from Heidegger precisely in his understanding of temporality. To simplify Stiegler's complex argument, his disagreement with Heidegger revolves around the different importance that the two philosophers give to the historical specificity of technology. Actually, Stiegler finds a contradiction in Heidegger's thought on technology and time - one that he also points out in *The Ister*. For Heidegger temporality is originally technical, since to be a temporal being - that is, to exist - one has to be in the world, which for Heidegger is fundamentally the world of tools – or of technology. Nevertheless, Heidegger believes that the most authentic temporality is experienced by human beings as a relation to death. As human beings, we are structurally ignorant of the time and place of our own death, and this relation to death plunges us into an 'absolute indetermination' (Stiegler 2003b: 159). We do not know the end of our life, both in the sense of its limit and of its meaning. In Stiegler's terms, the content of our life is determined only after our death, that is, 'too late', when we are not able to witness it. According to Heidegger, human beings try to flee the permanent anguish of the indeterminacy of their death, and ultimately of their future. This is what Heidegger names 'concern' - namely, the human beings' attempt to foresee their unforeseeable future, to make certain the uncertain, to calculate something that is not calculable. Technology is part of this process, precisely because it is a means of controlling the future. Every technical field, from weather forecasts to financial analysis, attempts such calculation. But any such attempt tends to obscure human beings' relation to death, and for this reason Stiegler ultimately finds Heidegger's argument inconsistent – since, on the one hand, Heidegger views temporality as originally technical, while on the other hand he believes that technology obscures 'authentic' temporality.

In sum, Stiegler's departure from Heidegger is based on Stiegler's own attention to the historic specificities of technology. I have mentioned earlier on how Poster similarly criticizes Heidegger's conception of technology for not being attentive to

historical specificity. Stiegler, however, pays close attention to the fact that human beings, as beings who deal with decision, 'are continuously called into question by the development of technics which overtake them' (162). Here Stiegler brings to the fore the problem of making decisions regarding technology that I confronted in my Introduction and with which I have also opened this chapter – that is, the problem of how to think technology in a politically meaningful way. The term 'overtaking' is deployed by Stiegler with reference, once again, to Leroi-Gourhan's and Gille's thought: 'people form technical objects, he argues, but these objects, because they themselves form a dynamic system, go on to overtake their makers' (162).[23] Technical objects form a 'system' because none of them is ever thinkable in isolation: a cassette – Stiegler exemplifies – is of no use without a tape recorder, which in turn is of no use without a microphone, electricity, and so on. Such systems are also dynamic: they change according to different tendencies that combine within society, and are negotiated through processes of decision.[24]

I want to highlight here how Stiegler's approach is extremely helpful in order to contextualize the necessity of making decisions about technology in the broadest possible perspective. In fact, such decisions do not just affect technology; they also change our experience of time, our modes of thought and, ultimately, our understanding of what it means to be human. On the one hand, if understood as originary, technology constitutes our sense of time – or, even better, we only gain a sense of time and memory, and therefore of who we are, through technology. On the other hand, technology as a system tends to overwhelm us, making it difficult to make decisions. Ultimately, we gain a sense of time through technology, and in turn every change in technology changes our sense of time - and this then changes the meaning that we give to the fact of being human. For this reason, as I argued in my Introduction, thinking technology in a politically effective and meaningful way involves much more than, for instance, discussing technology in a neutral and 'free' language, as Habermas would have it. Rather, it implies a radical problematization

[23] For the concept of technology overtaking its makers Stiegler relies on Marx as well as on Leroi-Gourhan, Gille and Simondon (Stiegler 2003b: 162; Leroi-Gourhan 1993; Gille 1986; Simondon 1989, 2001).

[24] The system of technical objects, as Stiegler points out, corresponds to Heidegger's 'system of reference' which constitutes the system of the world (Stiegler 2003b: 162). The concept of tendency is developed by Leroi-Gourhan drawing on Bergson's own notion of 'tendency' (Stiegler 2003b; Leroi-Gourhan 1993).

of the meaning of humanity. In fact, by making decisions on technology, we make decisions about what it means to be human. Ultimately, this is the sense of my affirmation that it is necessary to think technology *philosophically* in order to think it *politically*.

To conclude, I argue that Stiegler's rereading of Derrida calls for a concrete analysis of historically specific technologies while keeping open the significance of such an analysis for a radical rethinking of the relationship between technology and the human. It therefore enables me to regard software in its historical specificity without losing the possibility of investigating its relationship to 'originary technicity' and, ultimately, to the question of what it means to be human. But to what extent and in what way can such an investigation of software be pursued? An attempt to answer this question will be the focus of the next chapter.

2 Language, Writing and Code

Towards a Deconstructive Reading of Software

In Chapter One I pointed out the importance of the study of software for the understanding of the relationship between technology and the human. Drawing on Bernard Stiegler's work, I argued that software must be studied as a historically specific technology precisely in order to establish its significance for the radical rethinking of the above relationship. In this chapter I intend to investigate to what extent and in what way software can be explored in its historical specificity without overlooking its relevance for what Richard Beardsworth terms 'originary technicity', Jacques Derrida calls 'writing in general' and Stiegler describes as 'the originary prostheticity of the human'.[1]

In order to do so, let me go back for a moment to Jacques Derrida's exposition of the relationship between the study of technology and the concept of originary technicity. As I explained in Chapter One, according to Timothy Clark, Derrida can be considered one of the most important thinkers of originary technicity because he refuses to explain either technology or language in instrumental, functionalist terms,

[1] Let me recall part of my previous argument here. In Chapter One I showed how Western philosophical thought has excluded technology on the basis of its 'instrumentality' – that is, as a tool used to bring about certain ends. In the same way, philosophy has undervalued writing, also considered secondary to language and therefore instrumental - namely, a technology in the service of language. Thus, I turned to an alternative tradition of thought on technology (starting with Martin Heidegger and including Jacques Derrida and Bernard Stiegler, among others). Timothy Clark (2000) calls this the tradition of 'originary technicity' – a term he borrows from Richard Beardsworth (1996). This term assumes a paradoxical character only if one remains within the instrumental understanding of technology: if technology were instrumental, it could not be originary – that is, constitutive of the human. Thus, the concept of 'originary technicity' resists the utilitarian conception of technology. The thinkers of 'originary technicity' point out that technology is actually constitutive of philosophy, since, by providing the support for the inscription of memory, it allows for transcendence and therefore for thought.

and by doing so he resists the widespread denigration of the 'merely' technical in Western thought (Clark 2000: 240). As Clark also points out, Derrida's thinking on technology pursues the double strategy of recognizing originary technicity while at the same time revealing technology's complicity with metaphysics. Clark writes:

> Derrida's arguments on technology affirm an originary technicity or supplementarity both constituting the human and transgressing any would-be pure or essentialist distinctions between concepts of the human and the machine. At the same time ... [deconstruction] works to reveal and undo the profound complicity between metaphysical humanism and projects which idealize the technical as the correlate of a totally assured system of formal elements whose syntax or mechanics can be calculated – the notion of a technics which can be completely subordinated to logic.
>
> (Clark 2000: 248)

As the above passage shows, for Clark the first part of Derrida's double strategy consists in his conceptualization of originary technicity. Deconstruction 'upsets received concepts of the human and the technological by affirming their mutual constitutive relation or, paradoxically, their constitutive disjunction' (Clark 2000: 247). Neither term can be said to take precedence over the other. Thus, technology cannot be understood as a tool for the human; nor can the human be understood as an effect of technology. 'The identity of humanity – Clark specifies - is a differential relation between the human and technics, supplements and prostheses' (Clark 2000: 247).[2]

The second aspect of Derrida's double strategy is his questioning of technology and technicist thought, in order to reveal their complicity with metaphysical humanism, and particularly with the instrumental conception of technology that characterizes Western philosophy. For Clark instances of such complicity are cybernetics and

[2] As I showed in Chapter One, the argument for the originary technicity (or 'originary prostheticity') of the human also plays a key role in Bernard Stiegler's work (Stiegler 1998a; 1998b; 2001a; 2003a). This formulation emphasizes the co-emergence of humanity and technology and resists the Aristotelian view of technology as a mere tool.

artificial intelligence, 'or other kinds of formal systems modelling' (Clark 2000: 248).

In sum, Clark points out that Derrida views technology both as constitutive of the human and of knowledge, and yet complicit with the Western system of thought that for centuries has excluded technology from knowledge and has viewed the human as opposed to its tools. In Derrida's words:

> Computer technology, data banks, artificial intelligences, translating machines, and so forth, all these are constructed on the basis of that instrumental determination of a calculable language. Information does not inform merely by delivering an information content, it gives form, 'in-formiert', 'formiert zugleich'. It installs man in a form that permits him to ensure his mastery on earth and beyond.
>
> (Derrida 1983: 14)

In fact, the two parts of Derrida's strategy cannot be easily separated. It is precisely by unmasking and undoing – or, in Derrida's words, deconstructing - the complicity between metaphysical humanism and the 'idealization' of technology as something totally predictable ('calculable') that one can make originary technicity – that is, the constitutive relation between technology and the human - apparent. In fact, I want to argue that Derrida's double strategy helps me reformulating the question of how to study software as such: whether and in what way does software display a certain complicity with the instrumental understanding of technology? This question implies a second one – namely, to what extent, by undoing this complicity, light can be shed on the relationship between software (and more broadly technology) and the constitution of the human. Although these questions will find a definitive answer only in the following chapters, in order to set out the terms of the problem it is worth examining here at length Derrida's insight on new technologies and his clarification of how difficult it is to conceive them merely in terms of instrumentality.

Quite strikingly, for Derrida new technologies are already 'in deconstruction' in what he terms today's 'technological condition'. In his conversation with Bernard

Stiegler in *Echographies of Television* (Derrida and Stiegler 2002) Derrida awards this process of deconstruction quite a broad meaning. For him the acceleration of technological innovation in the contemporary world, coupled with the development of information and telecommunication technologies (and more broadly with the technologies of the global media system – which Derrida and Stiegler group under the rubric of 'tele-technologies'), constitute a 'practical deconstruction' (Derrida and Stiegler 2002: 45) of the traditional political concepts of the public, the state, the citizen, and ultimately of the instrumental conception of technology itself. On the one hand, in the contemporary world technological innovation is massively appropriated by multinational corporations and nation states, by means of their 'research and development' and 'defence' departments. Within this context, technological products become obsolete very quickly and technological innovations are constantly programmed to support such an economy of continual obsolescence. On the other hand, although programmed and neutralized as controlled 'development', technological innovation still gives rise to unforeseen effects. Derrida even propounds that the greater the attempt to control innovations, the more unforeseeable the future becomes. This second point is well exemplified for him by the relationships between telecommunications and the transformation of the public space. For instance, in *Echographies* he states:

> I believe that this technical transformation - of the telephone, of the fax machine, of television, e-mail and the Internet - will have done more for what is called 'democratization', even in countries in the East, than all the discourses on behalf of human rights, more than all the presentations of models in whose name this democratization was able to get started.
>
> (Derrida and Stiegler 2002: 71)

Here Derrida refers explicitly to the relations between telecommunications and the political transformations of the late 1980s and early 1990s in Eastern Europe. However, as Clark points out, Derrida does not see this process as the realization of liberal humanist values, but as 'an accelerating process of spectralization' (Clark 2000: 249). To understand this point better, it is important to examine what Derrida argues in relation to spectralization and tele-technologies in other parts of his work,

and most significantly in *Spectres of Marx* (1994). In this text Derrida emphasizes how older notions of the public sphere are being disrupted by tele-technologies and by the new rhythms of information and communication, and how the analysis of the public space today must take into account the role of new media and of their 'spectral effects' – that is, of 'the new speed of *apparition* ... of the simulacrum, the synthetic or prosthetic image, and the virtual event, cyberspace and surveillance, the control, appropriations, and speculations that today deploy unheard-of powers' (Derrida 1994: 54). For example, images and information are disseminated beyond the borders of territorially delimited communities. Media power makes contemporary politics dependent on the modes of its mass presentation. The main function of national governments increasingly involves managing the state in the interest of international capital. Broadly speaking, new technologies transform the relationship between the public and the private. This last point is also touched upon by Derrida in his essay of 1996 on psychoanalysis, entitled 'Archive Fever: A Freudian Impression' (1996), where he muses over some of the ways in which digital databases and electronic mail could have changed psychoanalysis, had they been available in the early days of the discipline. Electronic mail would have enabled extensive correspondence between psychoanalysts as well as an automatic archivization of such exchanges (therefore transforming psychoanalytic stored memory and – ultimately - knowledge). Even more broadly, electronic mail would have changed 'the entire public and private space of humanity' (Derrida 1996: 17). Although Derrida does not elaborate on this suggestion in 'Archive Fever', a number of scholars have commented extensively upon the transformation of the relationship between the private and the public brought about by new media. For instance, Mark Poster has convincingly argued that electronic surveillance and digital databases erase the dominion of the private (Poster 2006: 148). Joshua Meyrowitz has pointed out how television introduces public occasions into the intimate, private space of the living room (Meyrowitz 1985). Even more fittingly, Poster has also shown how the modern subject's sense of its exclusive awareness of its own thoughts is restructured by electronic communication (Poster 2006). However, what is particularly relevant in *Spectres of Marx* is Derrida's argument that the speed and pervasiveness of new media oblige us more than ever to think about the

virtualization of space and time, and prevent us from opposing an event to its representation, and 'real time' to 'deferred time' (Derrida 1994: 169).[3]

Ultimately, according to Derrida the unforeseen effects of tele-technologies deconstruct the perception of the human being as separate from his tools but also as a master of them. Clark explains:

> To conceive technology as a prosthesis that alters the very nature of its seeming user is to be reminded how technical inventions have always been in excess of their concepts, productive of unforeseeable transformations. This deconstruction of the Aristotelian system enables each invention to be seen as an irruption of the other, the unforeseen disrupting the very criteria in which it would have been captured.
>
> (Clark 2000: 249)

This passage clarifies that there are certain effects of technological innovation that are beyond ('in excess of') what can be expected within (and produced by means of) a procedural method. This leap beyond the programmable, which was once ascribed to 'genius' or 'inspiration' and therefore recuperated as part of the philosophy of subjectivity, is now the result of deconstruction as the bringing about of what cannot be calculated or programmed.[4] The most striking example of such unexpected consequences of technology is perhaps the Internet and its growth, out of a seminal military structure of the 1970s (the ARPANET), into an extended network whose expansion is only in part controllable. Poster (2006) points out that digital machines,

---

[3] One must be reminded at this point that in *Spectres of Marx* Derrida instantiates a profound connection between deconstruction and a commitment to justice and democracy. He insists that deconstruction is a positive intervention in the sphere of the political, and, while acknowledging that Marx 'is one of the rare thinkers of the past to have taken seriously ... the originary indissociability of technics and language' (Derrida 1994: 53) – and besides recognizing that we are all heirs of Marx to a certain extent, and indebted to the line of thought we call 'Marxism' – he also emphasizes that Marx could not experience tele-technologies as we can today (53). For Derrida the virtualization brought about by tele-technologies entails a rethinking of the public space, of 'live' transmission and of its relationship to what counts as an event. Such rethinking is the key to a new political thought which would entail the capacity of remaining open to the future – that is, to a future event that cannot be determined in advance (what Derrida terms 'a new international', 'the promise', or 'democracy to come'). Ultimately, according to Derrida tele-technologies require us to be able to think politics differently, to think 'another space for democracy' – namely, to remain open to the unexpected (possibly engendered by the unforeseen consequences of technologies), to the future, to 'the coming of the other' (169).

[4] Derrida makes such observations in relation to the romantic theory of genius in his essay 'Psyche: Inventions of the Other' (Derrida 1989: 58-60).

even though originally funded by the military and subsequently appropriated by corporations, have been implemented by academic researchers with the aim of transferring information quickly and efficiently - that is, without any noise. They have not been implemented to pay attention to 'who is authorized to speak, when, to whom, and what may be said on these occasions' (Poster 2006: 51). These structural aspects of technology ultimately enable technically noiseless channels of communication on which digital cultural objects - that is, texts, images and sound - circulate rather freely. Such circulation, in turn, endows global networking with the potential for contrasting hegemonic agglomerations of power.[5]

Derrida names such a leap beyond the programmable 'the coming of the other' (Derrida 1994: 55). In order to understand this concept better, one must recall that – as Clark explains - '[d]econstruction represents a rejection of the view, powerful since the seventeenth century, that humanity necessarily understands what it has made better than what it has not' (Clark 2000: 251). This observation posits a fundamental relationship between deconstruction and technology. Both Clark and Derrida acknowledge that technology has the capacity for interrupting reason and calculation, and for bringing forth something unexpected that cannot be totally understood within the existing conceptual framework. For instance, tele-technologies challenge the very concept of 'invention' as something that 'is assignable to a human subjectivity, individual or collective, responsible for the discovery or production of something new and publicly available' (Clark 2000: 251). The concept of invention must therefore be re-invented, and it exceeds the recognizability of the producer and of the product.

In sum, according to Derrida, today's technology is 'in deconstruction' because, with its capacity to generate unforeseen consequences, it challenges received notions of invention and instrumentality, and in so doing it challenges the (Aristotelian) notion of 'tool' within which technology itself has been constricted for centuries. This is also the reason why, as Clark makes explicit, technology cannot be simply an 'object' of deconstruction understood as a discrete methodology of analysis or

[5] The fact that many communication protocols *do* embed discriminations as to who speaks, when and to say what does not invalidate Poster's argument, since ultimately such technical characteristics do not introduce perceivable limitations on the circulation of cultural objects.

critique: '[t]echnology is "in deconstruction" as the condition of its being: for an originary prosthetic supplemementarity or originary technicity both opens and prevents realization of the Aristotelian and techno-scientific concepts of technology' (Clark 2000: 252). Accordingly, Derrida's argues that deconstruction is not a methodology, because it is not a set of immutable rules that can be applied to any object of analysis – since the very concepts of 'rule', of 'object' and of 'subject' of analysis themselves are subject to deconstruction too.

One must be reminded at this point that - as I clarified in the Introduction - 'deconstruction' is something that 'happens' within a conceptual system, rather than a methodology, and that any conceptual system is always in deconstruction, because it unavoidably reaches a point where it disassembles its own presuppositions. On the other hand, it is perfectly possible to remain oblivious to the permanent occurrence of deconstruction. Therefore, there is a need for us to actively 'perform' it - that is, to make its occurrence visible. In this sense deconstruction is also a productive process. Therefore, although what Derrida terms 'practical deconstruction' is already occurring in new technologies, there remains a need to make such occurrence visible through an active investigation of new technologies' 'complicity' with instrumentality. But to what extent and in what way does software, both 'open and prevent the realization of instrumentality' - as Clark would have it - that is, of the traditional philosophical understanding of technology as a tool? To what extent and in what way does software exceed its own instrumentality by giving rise to unforeseen consequences? And, most importantly, in what way can a deconstruction of software actually be performed?

In order to answer these questions, let me recall part of my argument from Chapter One regarding the connection between writing and technology on the one hand and instrumentality on the other. As we have seen, Western philosophical thought has devalued technology as a mere instrument. The traditional, Aristotelian view is that technology is extrinsic to human nature as a tool which is used to bring about certain ends. At the same time, philosophy has devalued writing, also considered secondary to language and therefore instrumental - namely, a technology in the service of language. Conversely, those thinkers who have distanced themselves from such instrumental understanding (mainly, but not exclusively, Heidegger, Stiegler,

Derrida, and André Leroi-Gourhan) have proposed a view of technology as a fundamental characteristic of human beings. They have actually suggested that philosophy has constituted itself precisely in relation (and in opposition) to technological knowledge, and they have pointed to the need for the radical rethinking of philosophy itself if an understanding of technology is to be made possible. In particular, they have emphasized that technology is actually constitutive of philosophy, since, by providing the support for the inscription of memory, it allows for transcendence and therefore for thought. According to Stiegler, transcendence is constituted through technology as the support for the inscription of memory (Stiegler 2003b; Hansen 2003; Beardsworth 1995). Stiegler actually establishes that technology and philosophy *(technê* and *epistêmê*) share a connection with writing, since, on the one hand, 'philosophical' problems (that is, problems that historically arise with Thales) fundamentally proceed from the appearance of a technique of writing, and, on the other, philosophy constitutes itself precisely as non-technical knowledge, excluding both technology and writing from its domain. Stiegler points out that the question of technology, considered as the object of repression, emerges through its denunciation by Plato: in *Phaedrus* Plato blames technology (identified with writing) for its power of forgetting (Stiegler 2003b: 154). Ultimately, in Chapter One I argued that there is a fundamental relation between technology and writing as both traditionally excluded by philosophy as 'instrumental' while at the same time constitutive of it. In the present chapter I want to establish to what extent this fundamental relation between technology and writing is relevant for the study of software – and, specifically, for the study of how software exceeds instrumentality.

In order to expand on this point, let me now turn to Katherine Hayles' recent conceptualization of the relationship that software entertains with Western thought on the one hand and with writing on the other. In her 2005 book, *My Mother Was A Computer*, Hayles reframes the problem of the relationship between software and instrumentality in terms of the relation between 'code' and 'metaphysics', at the same time introducing the concepts of language and writing as significant terms of this relation. As it will become clearer throughout this chapter, Hayles' understanding of 'code' is not identical to my understanding of 'software', although

they can be considered equivalent up to a certain point. For Hayles code is 'the language in which computation is carried out' (Hayles 2005: 17) – whereby 'computation' is defined as 'a process that starts with a parsimonious set of elements and a relatively small set of logical operations' which, instantiated into some kind of material substrate (such as a computer), can build up increasing levels of complexity, 'eventually arriving at complexity so deep, multilayered, and extensive as to simulate the most complex phenomena on earth, from turbulent flow and multiagent social systems to reasoning processes one might legitimately call thinking' (18).

Hayles examines the relations of code with metaphysics, where the latter is defined as a model 'for understanding truth statements' which is 'so woven into the structure of Western philosophy, science, and social structures that it continually imposes itself on language and, indeed, on thought itself, creeping back in at the very moment it seems to have been exorcised' (17). Hayles is particularly interested in 'the metaphysics of presence' which, after Derrida, she defines as the metaphysical yearning for the 'transcendental signified' – that is, for 'the manifestation of Being so potent that it needs no signifier to verify its authenticity' (17), or, in Derrida's terms, 'a *concept signified in and of itself*, a concept simply present for thought, independent of a relationship to language, that is of a relationship to a system of signifiers' (Derrida 2004: 19). I will return to this definition during my analysis of Hayles' argument. For now, suffice it to say that, when referring to the metaphysics of presence and to the transcendental signified, Hayles also uses the term 'ontology' as a synonym.

According to Hayles, code has a very loose relationship with metaphysics. She goes as far as to say that code 'inherit[s] little or no baggage from classical metaphysics', because it reduces ontological presuppositions to a minimum (Hayles 2005: 22). In order to understand this complex proposition, Hayles' argument needs to be examined more closely. The question of the relationship between metaphysics and code is considered by Hayles in the context of the broader analysis of the relations between speech, writing and code. She starts from the issue 'of how code can be related to theoretical frameworks for the legacy systems of speech and writing' (8).

She writes:

> Unnoticed by most, new languages are springing into existence, proliferating across the globe, mutating into new forms, and fading into obsolescence. Invented by humans, these languages are intended for the intelligent machines called computers. Programming languages and the code in which they are written complicate the linguistic situation as it has been theorized for 'natural' language, for code and language operate in significantly different ways.
>
> (Hayles 2005: 15)

As the above quotation makes clear, Hayles establishes a relationship between natural language on the one hand and programming languages and code on the other. She identifies a number of differences between the two: for instance, code is addressed both to humans and to computers, it is developed by small groups of technicians, and it is integrated into commercial production cycles and capitalist economics. She notices: '[a]lthough virtually everyone would agree that language has been decisive in determining the specificity of human cultures and, indeed, of the human species, the range, extent, and importance of code remain controversial' (15). On the one extreme of the spectrum there are people who see code as a niche artificial language aimed at computers. On the other extreme, we can find scientists such as Stephen Wolfram and Harold Morowitz, who advance the hypothesis that the whole universe is fundamentally computational, therefore viewing code as the language of nature – that is, as the *lingua franca* of all physical reality (5). Whether we view code as something very specific or utterly pervasive, Hayles urges for a theoretically sophisticated understanding of the interactions between code and language. She points out that, while there exists an enormous body of literature dealing with human languages and programming languages separately, scholarship engaging with the relationship between the two is comparatively limited. Yet interactions between language and code pervade our world. For example, human communication over the Internet involves natural language (e.g. when writing an email or using a chat line) as well as code (e.g. protocols of communications

between networked computers). The interactions between humans and computers are carried out through interfaces that rely on code but normally involve the display of messages in natural language. Interactions between natural language and code take place nearly every time computers are relied upon in order to perform everyday tasks, to the point that, according to Hayles, '[l]anguage alone is no longer the distinctive characteristic of technologically developed societies; rather, it is language plus code' (16). Hence the importance of developing a conceptual framework in which language and code can be thought together.

Up to this point it looks like Hayles' main preoccupation is the relationship between natural language and code. In fact, in the rest of the book she rapidly slips into a triadic model which involves speech, writing and code. Hayles identifies throughout history three main 'discourse systems' - namely, the system of speech, the system of writing, and the system of digital computer code – all of which are still at work in contemporary culture (16). According to Hayles, each of these systems is associated with a specific 'worldview' – that is, a particular set of premises and implications which can be detected, exemplarily and respectively, in the semiotic theory of Ferdinand de Saussure, in the grammatological thought of Jacques Derrida, and in the theories of a number of thinkers such as Wolfram and Morowitz, but also Ellen Ullman, Matthew Fuller, Matthew Kirschenbaum and Bruce Eckel, who have dealt with programming languages (Saussure 1988; Derrida 1976; Wolfram 2002; Morowitz 2002; Ullman 1997; Fuller 2003; Kirschenbaum 2002a, 2002b, 2004; Eckel 1995). Let me now examine Hayles' argument a little closer in order to evaluate its relevance for my investigation of the relationship between software and instrumentality. Hayles observes:

> From a systematic comparison of Saussure's semiotics, Derrida's grammatology, and programming languages, implications emerge that reveal the inadequacy of traditional ideas of signification for understanding the operations of code. ... The result is a significant shift in the plate tectonics of signification, with a consequent rethinking of the processes through which texts emerge. The point is not simply to jettison

> the worldviews of speech and writing – even if by some miraculous fiat this were possible – but rather to understand the processes of intermediation by which they are in active interplay with the worldview of code.
>
> (Hayles 2005: 8)

Thus, in order to understand the relationship between speech, writing and code it is crucial to understand in what way their respective worldviews conceptualize the process of signification. Before moving further into Hayles' argument, it is necessary to explain what she means by the term 'intermediation'. Hayles is interested in the interactions (which she describes as 'feedback loops') between language and code, humans and machines, old technologies and new.[6] Her goal is not to focus on any privileged point of view; for instance, she rejects the simplifying tendency to view the computer as 'the ultimate solvent that is dissolving all other media into itself' (39) that characterizes the work of media theorists such as Friedrich Kittler and Lev Manovich (Kittler 1999; Manovich 2002). On the contrary, she claims that social and cultural processes are much more complex than the convergence of all media into one via the process of digitisation - that is, the conversion of sound, image, text and their respective media into digital code. With the term 'intermediation' she points out that speech, language and code are in fact connected to each other through entangled and complex relations.

Hayles' concept of intermediation draws on what Jay Bolter and Richard Grusin have called 'remediation' – that is, 'the formal logic by which new media technologies refashion prior media forms' (Bolter and Grusin 2002: 273). According to this logic, older media (such as print) respond to the development of new digital media by seeking to reaffirm their status within a culture that challenges them. In their efforts to remake themselves and each other, both older and new media invoke a 'double logic' that involves the processes of 'immediacy' and 'hypermediacy' (5).

[6] A feedback loop is a process that circulates the output of a system back into the system as input. Although the idea of a system that governs itself by means of such process dates back to the Greeks, the concept of the feedback loop found its broader theoretical development and practical applicability in first-order cybernetics during the 1930s and 1940s. Hayles uses the concept both in relation to physical systems and to recursive conceptual structures (Hayles 2005: 246).

The former is the tendency of media to represent their productions as transparent and naturally accessible - for instance, live television and webcams aim at making viewers feel as if they were 'really there'. The latter is the opposite tendency of media to draw attention to their own strategies of representation - multiple video streams and split screen displays are also features of live television, and webcams can be embedded in hypermediated websites, where users can select from a 'jukebox' of webcam images to create their own display (Bolter and Grusin 2002: 6; Hayles 2005: 32). And yet, Hayles finds the term 'remediation' too restrictive for her purposes, since it refers specifically to those feedback loops that operate via the double logic of immediacy and hypermediacy and that assume as a starting point a certain locality and medium. With the more comprehensive term 'intermediation' she wants to emphasize the multiple causalities that influence interactions among media. Moreover, 'intermediation' also includes other kinds of interactions, such as those between humans and intelligent machines through mediating interfaces (Hayles 2005: 32 f.). However, the investigation of the processes of intermediation between speech, writing and code assumes an understanding of the differences and similarities between the three worldviews associated with them (8). According to Hayles, such differences become mostly apparent precisely in the relationship that these three worldviews entertain with metaphysics.

Thus, we are brought back to the initial question of the relation between code and metaphysics. We have already seen that for Hayles computation has a very specific relationship with metaphysics, and it must be understood in a much broader sense than as something that goes on within a computer. In this view, computation is not limited to digital machines and binary code, but can form the basis for the physical universe. She refers to Stephen Wolfram, who, in his book *A New Kind of Science*, makes the claim that the whole universe can be explained through processes of computation and according to the theory of cellular automata. For Wolfram, the complexity of the universe can be viewed as emerging through successive cycles of computation, starting from binary elements. Complexity emerges not from the

variety of the starting elements but from the number of iterations and from the unpredictability of their results (Wolfram 2002; Morowitz 2002).[7]

How does the worldview of computation position itself in relation to metaphysics then? Has we have seen, according to Hayles, it basically reduces ontological requirements to a minimum.

> Rather than an initial premise (such as God, an originary Logos, or the axioms of Euclidean geometry) out of which multiple entailments spin, computation requires only an elementary distinction between something and nothing (one and zero) and a small set of logical operations. The emphasis falls not on working out the logical entailments of the initial premises but on unpredictable surprises created by the simulation as it is computed.
>
> (Hayles 2005: 22f.)

The above passage clarifies that Hayles views the distinction between zero and one as minimizing the need for metaphysical foundations. For her the computational view of the universe requires no ontological foundations, other than the minimal presuppositions needed to set the system running. 'Far from presuming the "transcendental signified" that Derrida identifies as intrinsic to classical metaphysics', she remarks that 'computation privileges the emergence of complexity from simple elements and rules' (23).[8]

This last part of Hayles' argument seems to me rather problematic. In fact, when discussing the theory of computational universe, Hayles actually detects its

[7] Hayles does not embrace the theory of the computational universe as a plausible explanation of physical reality, but treats it as a productive hypothesis that provides her with a means to articulate the worldview associated with code (Hayles 2005: 30).

[8] For instance, Edward Fredkin's 'digital philosophy' implicitly positions itself as an answer to metaphysical questions – as Hayles remarks following Wolfram (2002). Drawing on the discrete nature of elementary particles, Fredkin understands the whole universe as digital – that is, as 'software running on an unfathomable digital computer' (Hayles 2005: 23). As he explicitly acknowledges, digital philosophy 'carries atomism to an extreme' (23).

fundamental ambiguity - that is, its uncertainty whether computation should be understood as a metaphor (however pervasive it is in our culture) or as having an 'ontological status as the mechanism generating the complexities of physical reality' (20).[9] She explicitly avoids taking sides on such a controversial issue (especially since the theory of the computational universe has not been proved). Rather, she accepts that such a question remains undecidable as of today and that the computational universe is able to function simultaneously as means and metaphor. She argues that, regarded as a culturally potent metaphor, computation can actually invest the social construction of reality, as is the case of the reorganization of the US military according to 'network-centric warfare'. The presupposition of information as a key military asset leads to the reorganization of the military as a mobile and flexible force and as 'a continuously adapting ecosystem' (21). Thus, Hayles continues:

> Anticipating a future in which code (a synecdoche for information) has become so fundamental that it may be regarded as ontological, these transformations take the computational vision and feed it back into the present to reorganize resources, institutional structures, and military capabilities. Even if code is not originally ontological, it becomes so through these recursive feedback loops.
>
> (Hayles 2005: 21f.)

In this – again debatable – statement Hayles seems to have moved away from the conception of information that she supported in her 1999 book, *How We Became Posthuman*. In that book she showed how Shannon and Weaver's understanding of information as a mathematical function independent of its material substrate was able to generate the possibility of ultimately thinking the human as a disembodied entity. In her most recent book, conversely, she analyzes the formal definition of computation in order to explore its potential to generate embodied social and cultural

[9] For instance, Wolfram's work shifts from regarding computation as a way to conceptualize complex system (as in the simulation of life processes by means of cellular automata running in a computer) to thinking computation as a process that 'actually generates reality' (Hayles 2005: 19).

(as well as physical) structures. In *How We Became Posthuman* she regarded Hans Moravec's desire to upload the human consciousness into a computer as emblematic of a disembodied posthumanity, which she argued against. In *My Mother Was A Computer* she embraces Richard Doyle's observation in *Wetware* (Doyle 2003) that the desire to upload one's consciousness into a computer (and thereby achieve immortality) has already provoked a new perception of one's self and of the others, or what Doyle calls a 'techno-social mutation' (Hayles 2005: 22).

What I want to emphasize here is that, ultimately, Hayles' theory of computation does not seem to break free from the dilemma of whether code has 'ontological status'. Even more importantly, I want to argue that she cannot get rid of it precisely because she conceptualizes code within a triadic model that sets speech, writing and code as three separate entities associated (as we have seen above) with three separate worldviews that, albeit in complex interactions (which she calls intermediations), can still be placed in a kind of temporal progression with respect to their relationship with metaphysics. I believe that this triadic model needs to be rethought in order to give account of the relationship between 'code' and 'metaphysics', as well as of the one between technology (and specifically software) and instrumentality. Let me now develop this point a little further by returning to the fact that Hayles associates a specific reference text - and a specific theory - with each of the three worldviews, which is presumed to give an account of that worldview's relationship with metaphysics.

According to Hayles, the worldview of computation shifts the locus of complexity from Logos (that is, from the ideal originary point that both exceeds and generates the world) to 'the labor of computation that again and again calculates differences to create complexity as an emergent property of computation' (41). Here Hayles acknowledges the importance of Derrida's deconstruction of metaphysics, and yet she claims that Derrida's grammatology locates complexity in the 'trace', and that therefore it cannot be relied upon in order to give an accurate account of the processes of computation (41). Actually for her the different locations of complexity in Saussurean linguistics, Derridean grammatology and computation 'have extensive

implications for their respective worldviews' (41). She starts from the examination of Ferdinand de Saussure's 1915 book, the *Course in General Linguistics*, which is generally considered the foundational text of modern linguistics (Saussure 1988). According to Saussure, the sign has no 'natural' or inevitable relation to that which it refers to – that is, in his own words, 'the linguistic sign is arbitrary'. It is for this reason that Saussure excludes hieroglyphic and idiomatic writing from his consideration (Hayles 2005: 42). He 'regards speech as the true locus of the language system (*la langue*) and writing as merely derivative of speech' (42), and explains:

> A language and its written form constitute two separate systems of signs. The sole reason for the existence of the latter is to present the former. The object of study of linguistics is not a combination of the written word and the spoken word. The spoken word alone constitutes that object.
>
> (Saussure 1988: 25 f.)

I have discussed Derrida's analysis of the subordination of writing to speech in Chapter One, and I will return to it later on in this chapter. For now, suffice it to say that Hayles follows Derrida's critique of such prioritization. Moreover, she points out how Saussure tends to underplay the role of 'material constraints' of any kind in the functioning of the sign (Hayles 2005: 42). Briefly put, Hayles' argument is that the arbitrariness of the linguistic sign is much less valid in code than in speech and writing. She emphasizes that material constraints play a productive role in computation, 'functioning to eliminate possible choices until only a few remain' (42). Saussure ignores such constraints, and views meaning as emerging only from differential relations between signs. As Jonathan Culler remarks, for Saussure signs are 'purely relational or differential entities' (Culler 1986: 33). On the contrary, for Hayles, although material constraints also work in speech and writing (for instance, English, and natural languages in general, do not have words of one hundred or more syllables, for the simple reason that it would be impossible to pronounce them), they are much more important in code. While Saussure's theory erases materiality, the development of computers has been characterized by a dramatic centrality of it -

Hayles claims, 'from John von Neumann in the 1950s agonizing over how to dissipate heat produced by vacuum tubes to present-day concerns that the limits of miniaturization are being approached with silicon-based chips' (Hayles 2005: 43). But the most significant example of the role that material constraints have played in computation is the shift from analog to digital computers. For Hayles one important instance of such shift is the Transistor to Transistor Logic (TTL). In principle – she explains - it would be possible to build an analog computer, but this would make error control far more complex. Simply put, in TTL chips the binary digit zero is represented by zero volt, and the binary digit one by five volts. 'If a voltage fluctuation creates a signal of .5 volts - Hayles clarifies, it is relatively easy to correct this voltage to zero, since .5 is much closer to zero than to five' (43). Error control is therefore quite straightforward and simple, compared to analog computers, where voltages vary continuously. Hayles concludes: '[f]or code, then, the assumption that the sign is arbitrary must be qualified by material constraints that limit the ranges within which signs can operate meaningfully and acquire significance. … In the worldview of code, materiality matters' (43).[10]

In sum, for Hayles code has a much stronger connection to materiality than speech and writing. Moreover, she claims that Derrida's grammatology does not take into account the role of materiality in the process of signification. This part of Hayles' argument deserves a careful analysis. In fact, not only I want to argue that Derrida's conception of writing is based on his recognition of the materiality of the sign. Furthermore, as I began to show in Chapter One, Derrida's grammatology has much wider implications than Hayles recognizes, since it constitutes a problematization of the whole of Western metaphysical thought. What I want to emphasize here is that actually Derrida's understanding of writing can lead to the questioning of Hayles' own triadic model as well as to a radical reframing of her 'ontological dilemma' and, even more importantly, to an understanding of 'writing' that is particularly helpful for the conceptualization of what I name 'software' (and Hayles names 'code'). In order to elucidate this point, and before turning to the direct examination of

[10] A thorough discussion of the role played by material constraints in programming (for instance in terms of time and memory resources) is provided by Jay Bolter in his book *Turing Man* (Bolter 1984).

Derrida's concept of writing, let me keep examining Hayles' analysis of code for a little while.

Her discussion of Saussure's semiotics leads Hayles to ask whether any processes of signification actually take place in computation, and, if so, what a sign in the Saussurean sense (that is, as the unity of a signifier and a signified) would become in code. She follows Kittler's observation from his essay 'There Is No Software' that ultimately everything in digital computers is reduced to changes in voltages (Kittler 1997), and advances the proposition that signifiers in code coincide with voltages, while signifieds are 'interpretations that other layers of code give these voltages' (Hayles 2005: 45). In other words, Hayles views the microcircuitry in a digital computer as a signifier, and the first layer of code (namely, sequences of binary digits) as its signified. In turn, a higher level of code treats such sequences of binary digits as a signifier and attributes a certain meaning to them, according to the rules of the programming language in which that code is written. Programming languages operating at still higher levels translate the lower levels of signification into commands that more closely resemble natural language. Hayles explains that '[t]he translation from binary code into high-level languages, and from high-level languages back into binary code, must happen every time commands are compiled or interpreted, for voltages and the bit stream formed from them are all the machine can understand' (45).

This passage contains a number of technicalities, including the distinction between those kinds of programming languages that are interpreted and those that are compiled. It is not important here to deal with all these aspects in detail. What is worth noting is that Hayles views the correspondence between circuitry and binary digits as a process of translation in which materiality plays a very important part. For her in the computer the chain of zeroes and ones eventually 'becomes' circuitry. She argues further:

> Hence the different levels of code consist of interlocking chains of signifiers and signifieds, with signifieds on one level becoming signifiers

> on another. Because all these operations depend on the ability of the machine to recognize the difference between one and zero, Saussure's premise that differences between signs make signification possible fits well within computer architecture.
>
> (Hayles 2005: 45)

That computation partly fits a Saussurean model should come as no surprise, since historically programming languages have been developed according to structuralist semiotics. As I will show in greater detail in Chapter Five, the modern theory of formal languages (which is concerned with the specification and manipulation of languages, be they natural languages such as English or programming languages such as Pascal) was originated mainly by the work of the American linguist and mathematician Noam Chomsky, who in the 1950s, at MIT, developed a model for the formal description of natural languages based on what he called 'replacement rules' and 'transformations' (Chomsky 1965; Tanenbaum 1999).[11]

However, the relevant point here is that Hayles invokes a Saussurean understanding of the sign throughout her analysis of code, while at the same time upholding a stronger account of materiality *within* the Saussurean framework. In order to give an account of the materiality of code, Hayles follows Alexander R. Galloway in understanding code as executable. According to Galloway this is the essential difference between code and any other language. This is also why code is 'so different from mere writing': '[c]ode is a language, but a very special kind of language. Code is the only language that is executable' (Galloway 2004: 165). Hayles relies on Galloway's statement to substantiate her assertion that code has a 'performative' nature, at the same time turning to John Austin's theory of performativity as a way to take materiality into account in code. To understand her

[11] In the 1950s Chomsky also produced a classification of formal languages and grammars. One particular category of grammars, which he called 'context-free', became central to software development in the 1960s. As I will show in Chapter Five, during the early 1960s computer scientists drew on Chomsky's description of natural languages in order to develop the programming language called ALGOL60. In this context, John Backus and Peter Naur proposed a notation that rapidly became widely used for the specification of the syntax of programming languages. This notation is known as the 'Backus-Naur formula (BNF)', and actually is a context-free grammar (Salomaa 1973; Fischer and LeBlanc 1988).

move, let me take a step behind and briefly examine the theory of speech acts as it was developed by John Austin from the 1930s onwards.

In 1955 Austin expounded his theory in a series of lectures he gave at Harvard University, subsequently published under the title *How to Do Things with Words* in 1962. Austin's theory was meant to oppose what he named the 'descriptive fallacy' – that is, the view that a declarative sentence always describes a certain state of affairs, and thus must be either true or false (Austin 1972: 56). Austin establishes the existence of declarative sentences that do not describe anything, and of which it makes no sense to ask whether they are true or false. He points out that the utterance of these sentences coincides instead with the 'doing' of an action that cannot be described as the action of saying something. The classical example given by Austin is the 'I do' uttered in a marriage ceremony. Such utterances he names 'performatives' (as distinct from 'constatives', which are descriptive). The difference lays in the fact that, while constatives can be true or false, performatives can only be 'happy' or 'unhappy'. A performative is happy depending on the conditions under which it is uttered: normally, there must be some accepted conventions regarding the performative that are understood by the participants in the conventional procedure, which in turn has to be carried out correctly and generate the expected social behaviour in the participant themselves (112). Quite clearly, the 'felicity conditions' of performatives unfold in the social realm. Accordingly, Hayles stresses that, while performative language causes changes only in the mind and behavior of people, code always has a very particular involvement with materiality, since 'it causes things to happen, which requires that it be executed as commands the machine can run' (Hayles 2005: 49). In other words, for her the materiality of code is more straightforward than the materiality of performative utterances. 'Code that runs on a machine is performative in a much stronger sense than that attributed to language' (50). In fact, while the performative force of language is 'tied to external changes through complex chains of mediation', by contrast

> code running in a digital computer causes changes in machine behaviour and, through networked ports and other interfaces, may initiate other

> changes, all implemented through transmission and execution of code. Although code originates with human writers and readers, once entered into the machine it has as its primary reader the machine itself. Before any screen display accessible to humans can be generated, the machine must first read the code and use its instruction to write messages humans can read. Regardless of what human think of a piece of code, the machine is the final arbiter of whether the code is intelligible. If the machine cannot read the code or if the program does not work properly, then the code must be changed and corrected before the machine can make things happen.
>
> (Hayles 2005: 50)

The above passage presents a number of interesting points. First of all, Hayles introduces the terms 'writing' and 'reading' in relation to code, without questioning them. But what would it actually mean for a computer to 'read' code? And what does it mean, exactly, for a human being to 'write' it? Moreover, while marking out the difference between performative language and performative code, the above passage also introduces a problem of agency – that is, it clarifies that, as far as code is concerned, computers are the arbiters of the competence of the utterance. Hayles already articulated this very point in her 1999 book, *How We Became Posthuman*:

> In natural languages, performative utterances operate in a symbolic realm, where they can make things happen because they refer to actions that are themselves symbolic constructions, actions such as getting married, opening meetings, or as Butler has argued, acquiring gender. Computational theory treats computer languages as if they were, in Austin's terms, performative utterances. Although material changes do take place when computers process code (magnetic polarities are changed on a disk), it is the act of attaching significance to these physical changes that constitutes computation as such. Thus the Universal Turing Machine, which establishes a theoretical basis for computation, is concerned not

> with how physical changes are accomplished but with what they signify once they are accomplished.
>
> (Hayles 1999: 274)

However careful these observations, I still find Hayles' understanding of materiality and of the relationship between materiality and code rather problematic. Importantly, she describes the materiality of code as more 'direct', more straightforward than the materiality of language and writing. This greater significance of materiality for code is related to its being performative 'in a stronger sense' than language. According to Hayles' rereading of Butler, the performativity of gender is exemplary of a 'mediated' performativity - a performativity that has effects mainly in the 'symbolic' realm. An analogous assimilation of Derrida and Butler's concepts of performativity as limited to the 'symbolic' or 'discursive' realm comes back from time to time in the theories of technology and gender (see for example Barad 2003; Sedgwick 2003). And yet I believe that here Hayles underplays the role of materiality in Butler's theory of performativity. As Butler herself remarked in her 1997 work, *Excitable Speech*, language can produce very powerful effects on the materiality of the human body – effect that Hayles would without doubt consider quite 'straightforward'. To give but an example, in her poignant analysis of hate speech, Butler argues that human beings can be injured by language, and that linguistic injuries such as racial invectives can produce physical symptoms such as blushing and rage (Butler 1997: 4). On the other hand, how can Hayles claim that code has a 'direct' relation with materiality? When stating that code causes changes in machine behaviour 'through networked ports and other interfaces', she actually acknowledges the mediated character of materiality in code.

I want to suggest here that the famous re-reading that Derrida gives of Austin's theory in his 1972 essay, 'Signature, Event, Context' (Derrida 1988) can be very helpful in order to think the materiality of code differently. In this work Derrida calls into question the traditional understanding of 'communication' as a vehicle that circulates an identifiable content – that is, a meaning. The meaning or content can be 'communicated' by different technical means – namely, 'by more powerful technical

mediations', for example by writing (Derrida 1988: 3), without being affected. This interpretation of writing is proper to philosophy, and views writing as something that simply 'extends' the domain of communication. In this perspective, writing is actually placed under the category of communication: it is the consequence of the human capability to communicate (4). Derrida exemplifies this understanding of writing through Condillac's *Essay on the Origin of Human Knowledge*. He focuses on the passage where Condillac states that

> "[m]en in a state of communicating their thoughts by means of sounds, felt the necessity of imagining new signs capable of perpetuating those thoughts and of making them *known* to persons who are *absent*". (I underscore this value of absence, which, if submitted to renewed questioning, will risk introducing a certain break in the homogeneity of the system).
>
> (Derrida 1988: 4)

According to Derrida, this notion of absence is not specific to writing; in fact, it is characteristic of every sign. He explains:

> In order for my 'written communication' to retain its function as writing, i.e., its readability, it must remain readable despite the absolute disappearance of any receiver, determined in general. My communication must be repeatable – iterable – in the absolute absence of the receiver or of any empirically determinable collectivity of receivers.
>
> (Derrida 1988: 7)

In other words, a piece of writing must be repeatable in order to function *as writing*. The capacity of writing to remain writing in the absence of its intended reader and/or producer – this iterability - constitutes the capacity of the written sign to break free from its context (that is, the 'empirically determinable collectivity' of receivers or producers). For Derrida this capacity of functioning after having been severed from their context pertains to every kind of sign. He states that '[e]very sign, linguistic or non-linguistic, spoken or written (in the current sense of this opposition), in a small

or large unit, can be *cited*, put between quotation marks; in so doing it can break with every given context, engendering an infinity of new contexts in a manner which is absolutely illimitable' (12). And again:

> A written sign, in the current meaning of this word, is a mark that subsists, one which does not exhaust itself in the moment of its inscription and which can give rise to an iteration in the absence and beyond the presence of the empirically determined subject who, in a given context, has emitted or produced it. This is what has enabled us, at least traditionally, to distinguish a 'written' from an 'oral' communication. ... At the same time, a written sign carries with it a force that breaks with its context, that is, with the collectivity of presences organizing the moment of its inscription.
>
> (Derrida 1988: 9)

At this point it becomes clear why for Derrida the structure of writing is the structure of every possible mark. The possibility of disengagement from a context and of citationality belongs to the structure of every mark, spoken or written – otherwise it could not function as a mark. 'What would a mark be that could not be cited?' Derrida asks (12).

It is from this perspective that Derrida approaches the problematic of the performative. Austin seems to consider the performative as something that does not describe something pre-existent (something outside of language and prior to it); rather, the performative 'produces or transforms a situation, it effects' (13). For this reason, Austin was obliged to free the analysis of the performative from what Derrida calls the 'authority of the truth value' and to substitute for it 'the value of force' (13). As we have seen, a performative cannot be true or false, but only felicitous or infelicitous. So, has Austin shattered the traditional concept of communication? For Derrida, he has not. In fact, Austin's performative always requires a value of context (14), that is the intentionality of the speaker, the communication of an intentional meaning. In Derrida's words, Austin has not 'interrogated infelicity as a law' (15). Derrida thus asks '[w]hat is a success when the possibility of infelicity ... continues to constitute its structure?' (15). In order to

prove this point, and to perform at the same time such an 'interrogation of infelicity as a law', Derrida analyses Austin's understanding of citationality. Austin excludes the possibility of a performative being quoted. He actually considers a quoted performative (such as the pronunciation of the nuptial 'I do' by an actor on stage) as 'abnormal, parasitic' (16). But – Derrida objects – all language is citational in order to function as such, and *especially* a performative. A performative is only possible if it is citational: how could the 'I do' function if it were not identifiable as the citation of the marriage formula? Derrida writes: '[w]hat Austin excludes as anomaly, exception, 'non-serious', *citation* (on stage, in a poem, or a soliloquy) is the determined modification of a general citationality – or rather, general iterability – without which there would not even be a 'successful' performative' (17).

I will return to Derrida's understanding of the materiality implicit in every sign later on in this chapter. However, it should already be quite clear here that, if analysed from this point of view, Hayles' distinction between the performativity of language and the performativity of code would not hold. In fact, for Derrida every process of signification requires the sense of the material persistence of the sign, and there is no opposition between performatives operating in 'the symbolic realm' (Hayles 1999: 274) and performatives that take place in computer processors.

Actually, Hayles' analysis of code brings back a certain separation between materiality and signification that I argue is untenable and not very helpful in understanding how code works. For instance, drawing on such a distinction, Hayles isolates two more characteristics of code: besides being material and executable, code is also hierarchical and discrete. As for the first point, the hierarchical structure of code seems to depend precisely on its proximity to materiality. The lower the level, the closer code comes to the simplicity of zeros and ones. Conversely, it has been the ability to build up from this simplified base that has enabled the creation of high-level programming languages.[12] Moreover, Hayles remarks, '[a]long with the hierarchical nature of code goes a dynamic of concealing and revealing that operates

[12] Starting from this point, Hayles makes also a more complex argument about electronic literature, whose literariness appears to be built on a binary basis. She deploys Saussure's distinction between the syntagmatic/paradigmatic axes in an original way in relation to narratives stored in digital computers (Hayles 2005: 53 f.).

in ways that have no parallels in speech and writing' (Hayles 2005: 54). Since computer languages become more similar to natural languages (actually, more English-like) as they move higher in the 'tower of languages' - an expression that Hayles borrows from Rita Raley (2003) - they tend to hide 'brute' lower levels. Such practice carries considerable advantages in terms of the easiness of programming. 'Knowing how to conceal code with which one is not immediately concerned is an essential practice in computer programming. ... At the same time, revealing code when it is appropriate or desired also bestows significant advantage' (Hayles 2005: 54). As for the benefits of the practice of concealing, Hayles gives the example of object-oriented languages. These languages bundle code within 'objects' (that is, within abstract entities that can be treated as building blocks in the practice of programming). Therefore, any object is totally autonomous from all the others, and the code within it can be altered without affecting other objects in the same domain.

I want to emphasize here that what Hayles observes for object-oriented languages is also true for a lot of older languages. For instance, since the beginning of structured programming, the creation of so-called 'routines' (that is, pieces of code that could be recalled and inserted at any time into other code) responded to the same exigency. The basic principle of classical 'modular' programming (which divides software into parts, each of which performs a specific function with no need to know how other parts internally work) is one of concealment. As I will show in Chapters Three and Four, one of the principles of 'good programming', according to Software Engineering literature, is precisely the concealment of unnecessary information through a practice commonly known as 'black-boxing'. This is actually a general strategy through which software operates. On the other hand, and in relation to the practice of 'revealing', Hayles gives the example of the 'reveal code' command in HTML documents, which 'allows users to see comments, formatting instructions, and other material that may illuminate the construction and intent of the work under study' (Hayles 2005: 54). Again, many other examples could be provided, since this is also a general feature of programming. For instance, when programmers run debug software, a general practice is to look into the lower level of code to detect 'bugs' (that is, errors and malfunctions) that could not be found out just looking at

higher-level code.[13] In general, it could be said that, without the practice of concealment, the technology of computing would never have moved on from the use of punched cards to the deployment of high-level programming languages. Therefore, a certain degree of concealment seems to be embedded in software. Nevertheless, in the next chapter I will show that this cannot be explained by means of different degrees of 'proximity to materiality'.

Also related to materiality is the fourth important characteristic of code that Hayles identifies, namely 'discreteness', or the very fact that code is digital. Simply put, digitization is the operation 'of making something discrete rather than continuous, that is, digital rather than analog' (56). For Hayles, crucially, digitization is a specific characteristic of code that can hardly be found in speech and writing, and is scarcely mentioned by Saussure or Derrida. Although she acknowledges the importance of the 'blank' in Derrida's grammatological thought, and agrees that spaces play an important role in the digitization of writing, she does not relate the blank to software. Hayles writes:

> The act of making discrete extends through multiple levels of scale, from the physical process of forming bit patterns up through a complex hierarchy in which programs are written to compile other programs. Understanding the practices through which this hierarchy is constructed, as well as the empowerments and limitations the hierarchy entails, is an important step in theorizing code in relation to speech and writing.
>
> (Hayles 2005: 56)

The question posed by Hayles is: how do practices of making discrete work in the digital computer? (56). Basically, the bit stream in the computer inner circuitry is formed from changing voltages channelled through logic gates. 'From the bit pattern bytes are formed, usually with each bite composed of eight digits – seven bits to

[13] Hayles (2005) offers an interesting discussion of the potential held by such practices of revealing and concealment for artistic exploration. It must also be noticed that Matthew Fuller (2003) builds a large part of his theory of 'critical software' on the concept of concealment. I will return to Fuller's argument later on in this chapter.

represent the ASCII code, and an empty one that can be assigned special significance' (56).[14] At each stage of digitisation technology can 'embody features that were once useful but have since become obsolete' (56). For example, the ASCII code contains one of such features: that is, an encoding on seven bits which was originally related to a seven-bit code for a bell ringing on a teletype. Albeit teletypes are no longer in use, and therefore such code has become obsolete, it remain there, since retrofitting the ASCII code to change it would require too much work. Therefore, the ASCII code contains 'a fossilized mark' of an extinct technology (57).[15]

Hayles' general argument is that in the progression from speech to writing to code, every successor regime introduces new features that we cannot find in the predecessor, and that this is the case for discreteness. She acknowledges that for Derrida spacing was what made writing not a simple transcription of speech but something that exceeded speech. In the same way – she states – code exceeds both speech and writing, and cannot be incapsulated in them (57). Another feature that cannot be found in speech or in writing is compiling – that is, the process of translation of high-level code into binary digits. Compilers are necessary if code has to become operative. Hayles relates such specificity both to the process of digitisation and to the fact that code implies a partnership between humans and machines:

> Compiling (and interpreting, for which similar arguments can be made) is part of the complex web of processes, events and interfaces that mediate between humans and machines, and its structure bespeaks the needs of both parties involved in the transaction. The importance of compiling (and interpreting) to digital technologies underscores the fact that new

[14] The ASCII code is one of the international standard methodologies for coding alphabetical characters and numbers in the computer memory. It takes into consideration 256 characters and expresses every single one of them through a unique sequence of eight bits. In practice, it associates a numerical value from 0 to 255 with each of the 256 characters, and represents this numerical value on 8 bits. The ASCII code is particularly easy to use and can be effortlessly memorized by programmers because the capital and minuscule letters and the decimal ciphers are coded in sequence: for instance, 'A' is coded with the value 56, 'B' with 66 and do on; 'a' is coded 97, 'b' 98 and so on; the cipher '0' is coded 48, '1' is coded 49 etc.

[15] Teletypes were based on a seven-bit pattern because that was the technology available at the time.

> emphases emerge with code that, although not unknown in speech and writing, operate in ways specific to networked and programmable media. At the heart of this difference is the need to mediate between the natural languages native to human intelligence and the binary code native to intelligent machines.
>
> (Hayles 2005: 59)

As a conclusion, Hayles suggests that the process of making discrete has 'ideological implications' (60). She draws here on Wendy Hui Kyong Chun's assertion that 'software is ideology' and on her Althusserian reading of desktop metaphors (Chun 2003). Drawing on Althusser understanding of ideology as the subject's imaginary relationship to his or her real conditions of existence, Chun claims that desktop metaphors such as folders and trash cans create an imaginary relationship of the user to 'the actual command of the core machine', that is, to the 'real' technical context within which the user's actions are actually given meaning and responded to. Following Chun, Hayles speaks of an 'interpolation of the user into the machinic system', that is of a disciplining of the user by the machine to become a certain kind of subject (Hayles 2005: 61).[16]

Hayles' argument is utterly accurate – nevertheless, I believe that it does not do justice to the practice of 'discreteness' taking place in code. I actually want to argue that what Hayles calls 'discreteness' can again be seen as a characteristic of *every* sign. For instance, the emergence of alphabetic writing can be seen as a process of 'making discrete' in Hayles' terms. As I showed in Chapter One following Derrida's rereading of Leroi-Gourhan's work, alphabetic writing is the result of a process of 'linearization' that transforms 'picto-ideography' – that is, the early form of graphism tightly associated with figurative art and independent from spoken language – into a sequence of phonetic symbols subordinated to spoken language

[16] Interestingly, here Hayles uses the term 'interpolation', supposedly meaning the 'insertion' of the user into the context of the machine. Fuller (2003) also addresses the incorporation of a model of the user in the computer in his proposal for a critical theory of software. I will return to this point later on in the chapter.

and to its linear temporality.[17] Before examining this important point further, let me recapitulate Hayles' argument so far.

In sum, Hayles basically depicts the relationship between 'code' and 'metaphysics' as a loose one, since code is characterized by the minimization of its ontological premises. And yet, she seems to continue interpreting code through the structuralistic version of linguistics (Saussure, Austin) that, albeit partially explaining the functioning of code, does nothing to further our understanding of how software exceeds instrumentality. Moreover, she establishes a relationship of 'intermediation' between code and writing, adding 'speech' as the third term of the triadic model that is supposed to account for the development of contemporary technology. In fact, she differentiates between language and writing on the one hand and code on the other by arguing that the latter has a stronger relation with materiality. In doing so, she seems to bring back the same distinction between the material and the symbolic that we have seen as the foundation of Western thought and of the philosophical (in Hayles' words, 'metaphysical') devaluation of technology (and writing).

Importantly, the tradition of originary technicity calls into question precisely such distinction between the material and the symbolic, thus also questioning the instrumental understanding of technology. What I want to suggest here is that, rather than rethinking the relationship between 'code' and 'metaphysics' (in Hayles' words) as a minimization of the ontological requirements of code - a conceptual move that ultimately seems to trap Hayles' argument in an ontological dilemma - it would be more productive to approach the problem from the point of view of

[17] Leroi-Gourhan views the emergence of alphabetic writing as associated with the technoeconomic development of the Mediterranean and European group of civilizations. At a certain point in time during this process writing became subordinated to spoken language. He writes: '[w]ritten language, phoneticized and linear in space, becomes completely subordinated to spoken language, which is phonetic and linear in time. The dualism between graphic and verbal disappears, and the whole of human linguistic apparatus becomes a single instrument for expressing and preserving thought – which itself is channelled increasingly toward reasoning' (Leroi-Gourhan 1993: 210). By becoming a means for the phonetic recording of speech, writing becomes a technology. As a tool, its efficiency becomes proportional to what Leroi-Gourhan views as a 'constriction' of its figurative force, pursued precisely through an increasing linearization of symbols. Leroi-Gourhan calls this process 'the adoption of a regimented form of writing' that opens the way 'to the unrestrained development of a technical utilitarianism' (212). Expanding on Leroi-Gourhan's view of phonetic writing as 'rooted in a past of nonlinear writing', and on the concept of the linearization of writing as the victory of 'the irreversible temporality of sound', Derrida relates the emergence of phonetic writing to a linear understanding of time and history (Derrida 1976: 85).

originary technicity. As I have shown at the beginning of this chapter, such an approach opens up the following questions: firstly, In what way does software both participate in instrumentality at the same time as exceeding it? and secondly, To what extent can the relationship between technology and writing (as both traditionally excluded by philosophy as 'instrumental' *and* constitutive of it) can help us answer the first question?

With regard to the relation between software and instrumentality, Hayles' argument goes to great lengths to clarify how software works, but – as I have remarked above - falls short of taking into consideration software's potentiality for producing unexpected consequences that go beyond its 'normal' functioning. However, as I have argued earlier on following Derrida, although programmed and neutralized as controlled 'development', technological innovation still gives rise to unforeseen effects – that is, to consequences that are beyond what can be expected within (and produced by means of) a procedural method. Examples of this vary from the relationship between telecommunications and the transformation of the public space to the unanticipated development of the Internet out of the military structure of the ARPANET into an extended network whose expansion is only in part controllable.[18] Thus, I want to suggest that the circumstances in which software does *not* function – better, in which it does not function *as expected* – could tell us more about software than those in which software 'works'. But in order to understand such circumstances a different approach is needed – namely, an approach that views malfunctions as revealing points (or points where the conceptual system underlying software is clarified) rather than just seeking to explain how software functions ordinarily.

If examined in relation to the second question – that is, what the relationship between writing and technology means in the framework of originary technicity - Hayles' argument clearly acknowledges the need for an understanding of the relationship between 'code' (which I still regard here as a synonym for 'software') and writing. However, her attempt to place code and writing in a unitary framework does not give an account of the role that materiality plays in writing itself, and

[18] In Chapter Three I will show how the attempt to control the unforeseeable consequences of software development motivated the foundation of the field of Software Engineering in the late 1960s.

ultimately in software. Let me start from this second point and draw on Derrida's grammatological thought in order to investigate the role of materiality in software a little further, thus clarifying my own understanding of the relationship between 'code' (or software) and writing. In turn, this analysis will prove useful for showing how software participates in *and* exceeds instrumentality. Moreover, it will also confirm that Hayles' call for a 'turn to materiality' (Hayles 1999), which has had great relevance in media studies, is in fact quite belated, and that, as I have already hinted above, such a call actually results from a misreading of the post-structuralist tradition Hayles draws on (but ultimately departs from).

As I showed in Chapter One, for Derrida the subordination of writing to speech (or seeing writing as the representation of speech) has meaning only within the system of Western metaphysics. According to Derrida, the premises of Western metaphysics have been inherited by human sciences as well, and particularly by linguistics. In Richard Beardsworth's terms, for Derrida, 'the theory of the sign is essentially metaphysical' (Beardsworth 1996: 7). In *Of Grammatology*, Derrida focuses on the deconstruction of linguistics from this viewpoint. More precisely, according to Derrida the deconstruction of linguistics, and of its central concept, the concept of the sign, is exemplary for the deconstruction of metaphysics. Moreover, for Derrida the sign is so important because it is exemplary of the metaphysical devaluation of materiality. As Richard Beardsworth points out in his 1996 book, *Derrida and the Political*, 'the very possibility of the sign is predicated on an opposition between that which is conveyed (the signified, the *logos*, the non-worldly) and the conveyor (the signifier, the worldly)' (Beardsworth 1996: 7). The signifier is a material entity, such as a sound or a graphic sign; the signified belongs to the realm of concepts. For Derrida this opposition is the foundation of all the other oppositions that characterize Western metaphysics (infinite/finite, soul/body, nature/law, universal/particular, etc.). For him deconstructing the sign is a fundamental and exemplary move precisely because 'metaphysics is derived from the domination of a particular relation between the ideal and the material which assumes definition in the concept of the "sign"'(7). The sign is central because it constitutes the foundation of the distinction between signifier and thing, a distinction which in turn is the basis of *episteme* and therefore of metaphysics. Again in Beardsworth's words,

'...metaphysics constitutes its oppositions (here: the non-worldly/worldly and the ideal/material) by expelling into one term of the opposition the very possibility of the condition of such oppositions' (8). But how is such an expulsion performed? This point deserves careful analysis through a close reading of Derrida's text. In particular, I want to examine Derrida's analysis of Ferdinand de Saussure's thought at some length.

Saussure (1988) argues that linguistics (as a science of language) must exclude from its objects of study the graphic sign. In fact, linguistics as a discipline is based for Saussure on the very separation of

> the abstract system of *langue* - a system of distinct signs corresponding to distinct ideas, true of all languages – from the empirical multiplicity of languages with their linguistic, physical and physiological variations. The abstraction depends in turn on a distinction between what is internal and essential, and what is external and accidental to the system of *langue*.
>
> (Beardsworth 1996: 8)

In *Of Grammatology* Derrida detects some fundamental contradictions in Saussure's view. First of all, for Saussure *langue* is underpinned by social conventions. Nevertheless, in spite of it being conventional - '*langue* is a pure institution' (Saussure 1988: 15) - *langue* for Saussure is based on the 'natural unity' between its two components – that is, meanings and what he calls sound-images (*l'image acoustique*). In other words, Saussure claims that there is a 'natural' correspondence between the signifier and its meaning, or signified. The phonetic pronunciation of a word is somehow more 'natural' than its written form. As Beardsworth points out, 'it is this natural order that allows Saussure to set linguistics up as a science' (Beardsworth 1996: 9). Furthermore, the natural relation between the signified and the phonic signifier determines the reduction (which is supposedly natural too) of written signs to 'tangible forms of' phonic signifiers, therefore 'secondary representations of them', and 'inessential to the object of linguistics' (9). In sum, Saussure seeks a foundation for linguistics in a supposed natural order of things.

According to Derrida, the exclusion of writing from linguistics must be viewed as an 'ethico-theoretical decision' (Derrida 1976: 61). Such a decision is masked under the apparent naturality of the object under consideration, but 'revealed by the obsessive insistence with which the founder of linguistics wishes to expel writing from the essence of language' (Beardsworth 1996: 9). Beardsworth claims that Saussure has an almost moralistic attitude in expelling writing from linguistics and in denying that writing allows the most important access to language (9). However, what is significant is not merely the devaluation of writing. Rather, such devaluation shows that linguistics has been founded on a double movement: firstly, the making of a decision, and secondly, the justification of such decision through the claim that it is natural – therefore, a disguise of the decision itself under the pretence of naturality.

For Derrida, both linguistics and philosophy are predicated on the normative exclusion of writing from truth. Once again, one must be reminded here of the exclusion of writing from knowledge in Plato. Derrida's rereading of Saussure is based precisely on the recognition of this exclusion. Furthermore, according to Beardsworth, in rereading Saussure Derrida recasts 'the terms in which *all institutional violence* is to be thought' (Beardsworth 1996: 10). To understand this formulation better let me now examine Saussure's text and the contradictions that Derrida detects in it in 'Language and grammatology' (Derrida 1976).

The first contradiction can be located in the natural hierarchy that Saussure poses between speech and writing. Beardsworth explains:

> For it is this notion of arbitrariness which makes of the sign an *institution*, that is, something that is not natural. To say therefore that there is a natural subordination of writing to speech – that writing is a secondary representation of a primary unit of sound and meaning – whilst propounding at the same time that all signs are arbitrary is contradictory.
>
> (Beardsworth 1996: 11).

Furthermore, Saussure explicitly states that the word is a pragmatic decision (Saussure 1988: 158), clearly admitting that to take the word as the minimum unit of

analysis is a decision made at the foundation of linguistics. Thus, he practically removes the objectivity of linguistics while instituting it as science. As we have seen, what Derrida terms 'an ethico-theoretical decision' is the movement that institutes the object of a 'science' but pretends to be natural, whereas (being a decision) it is not – and this is what Derrida means when he says that such a decision is 'violent'. Beardsworth comments: 'The irreducibility of a decision shows that the most innocent 'theorist' is always also a legislator and a policeman. It is in this sense that any statement is a judgement which carries "political" force' (Beardsworth 1996: 12). This passage clarifies very well why every theory is political. There is always violence in theory, and every theorist is a legislator in so far as every theory or discipline needs to police its boundaries. Derrida is not suggesting that we avoid decisions, or that all decisions are equally violent. The point is rather that a decision is always needed, but not all decisions are the same: some of them recognize their legislative force, while others disguise it under a claim to naturality (affirming that they are 'objective science'). Moreover, 'the acknowledgement of the prescriptive force of one's statements' may make one more ready to transform a disciplinary field, given that the field is not a natural representation of a pre-existing 'real' (12). This is what Derrida terms the argument of a 'lesser violence' in 'a general economy of violence' (12). 'Lesser violence', at the level of theory, means acknowledging the normative force of one's decisions – that is, the fact that such decisions shape the field, the theory, the discipline. Through such recognition one may become more ready to change the field itself. As I emphasized in the Introduction, this point has been developed by Gary Hall (2002) in his analysis of cultural studies as a field particularly attentive to the institutional forces that shape academic knowledge. Hall also suggests that cultural studies should pursue a tighter connection with deconstruction in order to strengthen such an awareness. Furthermore, drawing on Derrida, Hall insists that technology influences disciplinarity and in particular that new technologies are changing the very nature and content of academic knowledge (Derrida 1996; Hall 2002). This is why in the Introduction I argued that a deconstructive study of software can bring about a significant reconsideration of the boundaries and contents of media and cultural studies.

However, according to Derrida, the force of Saussure's decision is revealed in what it suppresses, because what is suppressed returns as a contradiction, and causes either repeated acts of violence - that is, the policing of the field - or 'a radical redescription of the tensions structuring Saussure's legislative decision' (Beardsworth 1996: 12). Derrida accomplishes such redescription in two steps.[19] The first step is the famous generalization of writing made by Derrida, which actually finds its basis in Saussure's own theory. In fact, Saussure actually argues that writing covers the whole field of linguistics, since the sign in itself is 'immotivated', and writing is exemplarily immotivated. In other words, writing, being entirely conventional, perfectly exemplifies the conventionality of language – or 'all speech is already writing by its being immotivated' (13). For Derrida writing (the *graphie*) 'implies the framework of the *instituted trace*, as the possibility common to all systems of signification' (13). The concept of 'instituted trace' is very important here, since it represents the possibility of making conceptual distinctions. Thus, it represents the moment that precedes the opposition between nature and convention, allowing for the very possibility of their separation. The concept of the instituted trace takes into account and comprehends Saussure's act of foundation of linguistics as a discipline, with its refusal to include writing. As I have explained above, there is a strong relationship between the 'trace' and disciplinarity. The 'trace' accounts for the foundation of a disciplinary space with its constitutive exclusions, as well as for the return of that which is excluded within the disciplinary space (13).

A further step on the investigation of what Derrida calls the 'economy of violence' – a step that corresponds to another contradiction in Saussure - concerns the *phone,* and is the most important reinscription of Saussure made by Derrida. Here Derrida confronts the problem of materiality straightforwardly. He reads Saussure against himself once again, this time in order to show that 'both philosophy and linguistics are derivatives of a movement which constitutes them, but which they disavow in order to appear as such' (14). His argument turns around the difference that Saussure recognizes between phonemes and their 'sonorous concretization' - that is, between the 'sound-image' and its materialization (14). Such difference allows, for instance, for multiple pronunciations of the same phoneme which are at the same time

[19] I am still following here Beardsworth's reading of Derrida in *Derrida and the Political* (1996).

recognized as pronounciations of the same phoneme. The pivotal passage is Saussure's use of writing as a metaphor for the reduction of the phonic substance. Let me read this passage a little closer.

Saussure uses writing as an example for phonetics, and states that the sign is arbitrary, negative and differential - that is, it functions only through (reciprocal) opposition, and the means by which it is produced are unimportant (Saussure 1988: 165f.). There is a difference between the materialization of each phoneme and the acoustic-sound 'which they presuppose in order to be recognized as such, whatever the form of their materialization' (Beardsworth 1996: 15). This difference is transformed, in Platonism, into the difference between the transcendental and the empirical, ideal/material, infinite/finite, primary/secondary. This very difference constitutes consciousness and founds the possibility of recognizing things. One must be reminded at this point of Derrida's understanding of consciousness. As I showed in Chapter One, Derrida's move is informed by Husserl.[20] Husserlian phenomenology separates 'ideal objects – attained through what Husserl calls the phenomenological reduction (or *epokhe*) of facticity (the word with all its variations and contingencies) – and, on the other hand, the world in its difference and empiricity' (16). In *The Origin of Geometry* Husserl derives the possibility of the phenomenological reduction from writing. For him writing constitutes ideal objects. The condition of their ideality is 'their repetition through time and space', which in turns 'depends on their inscription on a support which transcends the empirical context' (16). Beardsworth explains:

> The very support that allows for transcendence form the material world is itself material, necessarily restricting the purity of the transcendence from the material that is aimed at. Conversely, such repetition is not possible unless the difference of each inscription re-marks the inscription, just as the concrete letter 't' is re-marked in order to be recognized as such by its 'acoustic image'. This analogy reveals that, if writing constitutes

[20] Beardsworth points out how Derrida radicalised Husserl in his introduction to *The Origin of Geometry* and in *Speech and Phenomena* (Derrida 1961, 1967c). For this reason, according to Beardsworth, Derrida is so attentive to Saussure's 'reduction of the phonic substance of the sign' (15). Beardsworth also highlights how the importance of Derrida's understanding of the reduction of phonic substance has been generally underestimated.

> transcendence, arche-writing comprehends the very process of constitution.
>
> (Beardsworth 1996: 16)[21]

As I showed in Chapter One, according to Derrida's radicalisation of Husserlian phenomenology, the transcendental is always impure, always already constituted through materiality (the empirical), because the condition of consciousness is repetition. But what are the consequences for the conceptualisation of writing? The main point here – and one of pivotal importance for my investigation of software – is that writing and the material have never been opposed. Writing *is* material because materiality is the condition for writing itself, and for signification. Let me now investigate this point a little further.

The difference examined by Derrida is between an empirical *t* and our ability to recognize it as an instance of the letter *t*. Derrida notes that the sound-image 'is not a phonic sound object but the "difference" of each of its concretisations, that is to say it is the possibility or schema of each of its materializations' (16). In other words, to make the phoneme recognizable we need to be able to relate it to other phonemes, but this relation is only possible through its inscription in the empirical; therefore, the transcendence of an empirical sound is possible only via the empirical repetition of it, which needs to be transcended to make it recognizable. The phenomenological reduction of materiality makes the materialization of the 'sound-image' possible. Derrida writes:

> The sound-image is the structure of the appearance of the sound which is anything but the sound appearing ... not the sound being heard but the being-heard of the sound. Being-heard is structurally phenomenal ... One can only divide this... by phenomenological reduction... [which] is indispensable to all analyses of being heard, whether they be inspired by linguistic, psychoanalytic, or other preoccupations.
>
> (Derrida 1976: 93)

[21] As a consequence, 'writing has always been the irreducible site of metaphoricity - whatever the tradition (Arab, Christian, Greek, Jewish)' (Beardsworth 1996: 16).

According to Derrida, the trace, or *différance* is the 'being imprinted of the print' - *être imprimé de l'empreinte* (Derrida 1976: 92) - again, separable only through phenomenological reduction of materiality. This distinction allows for the articulation of difference (as such) and for consciousness. It both accounts for and exceeds (the logic of) metaphysics (Beardsworth 1996: 17). For Derrida: 'the trace is not more ideal than real... it is anterior to the distinction' (Derrida 1976: 95). Imprint is irreducible (and this irreducibility is devised by Derrida through a radicalisation of Saussure's sound-image) to 'either traditional philosophical analysis or to any analysis such as that of linguistics which presumes to supersede the originary transcendental thrust of philosophy' (Beardsworth 1996: 17) The graphic or phonic sign is marginal, what is important is the 'middle ground' we reach. Beardsworth explains: 'neither suspended in the transcendental nor rooted in the empirical, neither in philosophy nor in any empirical negotiation of the world that refuses to pass through the transcendental. The refuse to pass through the transcendental condemns one to a description of the fact of difference which is unable to take into proper account the necessity and economy of violence, its "genealogy". It thereby repeats 'the naïve violence particular to the oppositional axiomatic of metaphysics' (17).

For Derrida the generalization of writing is confirmed by the analogy with writing that Saussure makes precisely when bracketing the material. This also allows him to confirm that 'arche-writing, as an *originary structure of repetition*, constitutes the structure of the "instituted trace" which comprehends the foundation, exclusion and contradiction of (the history of) linguistics' (17). To quote Beardsworth again:

> [f]or a *t* to have identity as a *t*, it must be repeated. There can be no identity without repetition; and yet, this very repetition puts in question the identity which it procures, since repetition is always made in difference. Absolute repetition is impossible in its possibility, for there can be no repetition without difference. The concepts of repetition and difference form the precipitate of the metaphysical dissolution of an originary aporetic structure of repetition which Derrida calls 'arche-writing' or 'trace'.
>
> (Beardsworth 1996: 17)

This passage clarifies how the structure of repetition is at the basis of the process of signification. The mark is subject to the law of repetition in difference: the opposition between speech and writing is, for Derrida, a determining one in metaphysics. 'He thus traces the law of this repetition, as well as the metaphysical decision to transform it into an opposition, through the linguistic mark' (18). This characterizes not only the linguistic mark, but *all* marks – that is, all marks are only possible within this process of idealization'. In Beardsworth's words:

> 'Arche-writing' brings together, therefore, the analysis of originary violence specific to the elaboration of the trace with the simultaneous reinscription by Derrida of the opposition between the transcendental and the empirical. In other words, it brings together Derrida's analysis of the institution (here, of linguistics) with his renegotiation of the frontiers between philosophy and the empirical sciences. Indeed, the one analysis cannot be separated from the other. The method of deconstruction constitutes from the beginning both a reinscription of the empirico-transcendental difference and an analysis of the irreducibility of violence in any mark.
>
> (Beardsworth 1996: 18)

In sum, for Derrida we need to have a sense of writing in order to gain a sense of orality. 'Writing' then takes precedence over orality not because writing historically existed before language, but because we must have a sense of the permanence of a linguistic mark in order to recognise it and to identify it. Ultimately, the sense of writing is necessary for signification to take place. But we can have a sense of the permanence of a mark only if we have a sense of its inscription, of its being embodied in a material surface. In other words, although we recognize the written form of a grapheme (let's say 't') only by abstracting it from all the possible empirical forms a 't' can take in writing, nonetheless we need such an empirical inscription to make this recognition possible. This is what Derrida means when he says that 'the transcendental' is always impure, always already contaminated by 'the empirical'. In other words, language itself is material for Derrida; it needs materiality (better: it needs the possibility of 'inscription') to function *as* language.

This interpretation contrasts with the consolidated (Anglo-American) reception of 'post-structuralism' and of Derrida's thought as unaware of the material aspects of culture, society, economics and politics (according to Derrida's famous statement that 'there is nothing outside the text'). On the contrary, what I want to emphasize here is that textuality and materiality are not opposed. There is no actual need – as it is often claimed – to 'go back to materiality' after the 'linguistic turn' in cultural studies, because materiality has always been there. Writing is material because materiality is the condition for writing itself, and for signification.

If materiality is the condition for signification, then every code is material. More precisely: the condition for code to function is the possibility of inscription. This is not the same as saying that software always has material *and* semiotic characteristics, or that it involves physical apparatuses as well as information, microcircuits and Boolean logics, the social and the conceptual. Of course this is all true, but what a material understanding of software means in addition is that software can function only through materiality - not because it has to run on a processor, nor because there are economic forces behind it, but because, as every other code, it functions only through materiality, since materiality is what constitutes signs (an therefore codes). Moreover, writing is based on the very same possibility of material inscription, and the fact that it has been posited in a significant relation with software by Software Engineering should come as no surprise at this point. Paraphrasing Bruno Latour, it might be said that 'we have never been immaterial'.[22]

But if every code is material, and if the material structure of the mark is at work everywhere, how are we supposed to study software as a historically specific technology? Two questions resurface here – namely, Stielger's question regarding the relationship between originary technicity and historically specific technologies (how is one supposed to distinguish software from other historically specific technologies in a way that is meaningful for understanding the relationship between technology and the human?) and Hayles' question on the relationship between code,

[22] The reference is here to Bruno Latour's famous book entitled *We Have Never Been Modern* (Latour 1993).

writing and language (to what extent and in what way can software be distinguished from other historically specific material inscriptions?).

At this point I would risk a proposition: there is no general approach to software that can establish once and forever how software works and what place it occupies in relation to 'writing' and 'language'. As I pointed out in the Introduction, software has never been univocally defined by any disciplinary field (including technical ones), and it takes different forms in different contexts. The definition of software that constitutes the conceptual foundation of Software Engineering – that is, software as the totality of all computer programs as well as all the written texts related to computer programs – is a particularly interesting starting point for the investigation of software because it establishes a very strong relationship between software and writing. Actually, in Software Engineering software is defined *as* a form of writing.[23] The essential move that such a definition allows me to make is to reformulate the question regarding the specificity of software as such: how does 'software' emerge as a historically specific technology in the discourses and practices of Software Engineering and in what relationship with 'writing' and 'language'? Although this question will find an answer only in the following chapters, I want to suggest here that the specificity of software as it is conceptualized in the disciplinary field of Software Engineering resides precisely in the relationship it entertains with writing – or better, with a historically specific form of writing. It is in this relationship that software – as Gary Hall would have it - finds its 'singularity'. In Chapter Three I will show how 'software', 'writing' and 'code' emerge together - and actually constitute each other as constantly shifting terms - in the context of Software Engineering at the beginning of the 1960s. However, what I want to emphasize here is that there is no 'writing' prior to 'software' – that is, there is no *such* writing as the one that emerges *in* and *with* Software Engineering. Such kind of writing emerges *only there*, and only in relation with software and code. One cannot exist without the others. Although this co-

[23] Software Engineering describes software development as an advanced writing technique that translates a text or a group of texts written in natural languages (namely, the requirements specifications of the software 'system') into a binary text or group of texts (the executable computer programs), through a step-by-step process of gradual refinement (Brooks, 1987; Humphrey, 1989; Sommerville, 1995). For instance: 'software engineers model parts of the real world in software. These models are large, abstract and complex so they must be made visible in documents such as system designs, user manuals, and so on. Producing these documents is as much part of the software engineering process as programming' (Sommerville, 1995: 4).

emergence constitutes the historical specificity of Software Engineering, it cannot be said that it also constitutes the specificity of 'all software', since the definition of software varies in time and space. In Chapter Three I will also argue that there is actually no general re- (or inter-) mediation between software, writing and code. Rather, a co-constitution of software, writing and code can be identified *in* and *as* Software Engineering. The singularity of 'writing software' – as a 'singular' practice distinct from other kinds of 'singular' practices, such as 'writing literature', or 'writing electronic literature' - is precisely this: that it emerges in relation to software and code in the (again, 'singular') context of Software Engineering.

In what way is one supposed to investigate this singular process of co-constitution then? In the Introduction I have already argued for a deconstructive reading of 'software' – that is (once again, according to the definition of software proposed by Software Engineering) of computer programs *and* of the whole of technical literature related to computer programs.[24] Here I want to emphasize once more that my strategy of making such texts 'legible' is not equivalent to a 'direct observation' of software or code. As I explained in the Introduction, I intend to take away all the implications of 'directness' that the concept of 'demystifying' or 'engaging with' software - or better, with the co-emergence of software, writing and code in Software Engineering - may bring with itself. This awareness of the difficulties of a close, even intimate, engagement with software does not hinder the critical and political potential of such engagement. On the contrary, I want to argue that an understanding of software that gives an account of its own 'ethico-theoretical decisions' (in Derrida's terms) responds to the need, propounded for instance by Fuller, to formulate a critical theory of software. In order to clarify this point and to explain in what way a deconstructive reading of software must be performed - and how it differs from the accounts of software proposed in the context of 'software studies', including those advanced by Hayles and Galloway - let me now examine Fuller's reflection on the political relevance of the critique of software in his book of 2003 entitled *Behind the Blip*.

[24] Importantly, the concept of 'software' supported by Software Engineering includes the whole of the technical literature that constitutes Software Engineering itself.

Fuller focuses on what he considers the few existent strands of 'critical approaches to software' (Fuller 2003: 10). Firstly, he discusses the group of technologies that go under the name of Human-Computer Interface (HCI) – that is, technologies intended to facilitate the use of computers by human beings. For him the interest of HCI resides in the fact that the interface is 'the point at which the machinations of the computer are compelled to make themselves available in one way or another to a user' (Fuller 2003: 12). Yet, Fuller criticizes the narrowness of the model of the user embedded in HCI, which he defines as 'functionalist' (13). A telling example is for him the human interface of a real-time system: a pilot can drop bombs or a stockbroker can move funds efficiently through a combination of ergonomics and information-design that makes reaction times measurable - where reaction time is defined as 'the number of interactive steps from task identification to task execution' (Fuller 2003: 13). According to Fuller, such interface is 'functionalist' because it relies on a model of the user conceived in terms of tasks and of quantifiable efficiency. Ambivalently, he recognizes that the 'idealization of the human' implicit in HCI 'nevertheless can latch onto flesh' (13) while at the same time making the real user disappear into such an idealization. Fuller calls this 'the fatal endpoint' of standard HCI: '[i]t empowers users by modelling them, and in doing so effects their disappearance, their incorporation into its models' (13). Of course there are 'human-centred' variants of HCI design, but the name 'human-centred' is itself flawed, because it shows that 'there is still a model of the human – what constitutes it, how it must be interfaced – being imposed here' (13). What Fuller suggests is a change in the focus of HCI that could be pursued through a shift from the model of the individualised user typical of standard HCI toward different approaches such as Participatory Design – where users provide continuous feedback to programmers in a process of cooperative design. However, according to Fuller the approach of HCI is too characterized by 'functionalism' to be genuinely critical – that is, however 'human-centred', computer interfaces are designed on the basis of an ideal model of the user, to whom they provide predefined functions.

However accurate, Fuller's argument raises a number of problems. Not only, if viewed in the perspective of originary technicity – that is, keeping in mind the co-emergence of the human and the technical - the claim that one interface is 'more

human' than the other has no actual significance.[25] Even more importantly, Fuller deploys the concept of 'functionalism' in order to evaluate the critical potential of HCI. To understand this conceptual move better, let me examine the second group of critical approaches to software presented by Fuller – namely, programmers' own accounts of their practice (for instance, Ullmann 1997). Fuller views these accounts as descriptions of 'the interrelations of programming with other formations – cultural, social, aesthetic. These are drives that are built into and compose software rather than use it as a neutral tool' (Fuller 2003: 15). The main point here is that for Fuller the accounts of programming are at odds with 'the idealist tendencies in computing' (15) – that is, again, with the 'functionalist' approach of software design. In other words, according to Fuller the technical literature concerning software has a normative approach (it describes *how to* design software), while programmer's own accounts have a more realistic approach (they describe how software *is actually* designed). Even more importantly, in technical literature software is viewed as instrumental ('a neutral tool'). In order to find an alternative to the instrumental conception of technology, for Fuller it is necessary to turn to programmers' accounts of their practice, which take into accounts 'other formations' (the cultural, the social, the aesthetic). I can only suppose here that Fuller would view the technical literature of Software Engineering as an example of idealized description. Actually, in their most stabilized form, Software Engineering manuals basically describe how a software project should be managed – that is, what tasks must be accomplished (and in what sequence) in order to obtain a 'good' software product. In fact, what Fuller calls 'functionalism' is a general characteristic of technical literature – from the theory of programming languages to the most recent works by open source programmers, and including the technical literature of Software Engineering (see, for instance, Raymond 2000). However, my aim in Chapters Three and Four will be to demonstrate how technical literature can be read in a non-functionalist way – or, as I explained in the Introduction, in a deconstructive way – that is, in order to unmask how its conceptual system works. Such a reading will also make apparent how the instrumental understanding of software in technical literature is much more controversial and unstable than Fuller

[25] In Chapter Three I will show that what Fuller calls 'the user' is actually the way in which the human constitutes itself in relation with software.

assumes it to be. In fact, it will become clear that Software Engineering literature opens up the possibility of an instrumental understanding of software – an understanding that is however always at risk of coming undone because of its capacity to produce unforeseen consequences.

Of course, Fuller's strategy of opposing programmers' own accounts (how things 'really' are) to prescriptive technical literature (how things should be) can also be pursued. Nevertheless, I suggest that a much more interesting approach would involve showing how technical literature has been constituted as such – and thus, ultimately, how software has become what it is. Once again, Software Engineering constitutes an extremely promising starting point, since the moment of its constitution as a technical discipline is widely documented. In Chapter Three I will examine the technical literature produced in the early days of Software Engineering and I will discuss the choices that were made in 1968, when the first ever conference on Software Engineering was held in Germany. Importantly, the professionals and scholars who made such choices were very aware that the concept of 'software' was extremely controversial. They struggled to establish and maintain the boundaries of their discipline and often could barely reach an agreement on many of the foundational issues of Software Engineering. They were also able to produce an account of those initial moments, which can be read deconstructively in order to uncover the points of opacity that were incorporated in Software Engineering in order to establish it as a discipline. Rather than opposing the prescriptive approach of those early texts of Software Engineering to an hypothetical account of 'how software was actually designed', a deconstructive reading aims at 'undoing, decomposing, desedimenting' the conceptual system underlying Software Engineering, not in order to destroy it but in order to understand how it has been constituted (Derrida, 1985). As I explained in the Introduction, in every conceptual system we can detect a concept that is actually unthinkable within the conceptual structure of the system itself – therefore, it has to be excluded by the system, or, rather, it must remain unthought to allow the system to exist. Thus, in Chapter Three I will investigate what needs to remain unthought in order to establish Software Engineering as a discipline and to prevent its conceptual system from undoing itself. In a Derridean sense, such a deconstructive reading will show the 'violent' constitution of the discipline of Software Engineering.

In order to emphasize the political relevance of this non-functionalist reading, let me now discuss the conclusion of Fuller's argument. Fuller views software as a participial object: 'whilst one might deal with a particular object, it must always be understood not as something static, although it might never change, but to be operating in participial terms' (18). The term 'participial' comes from Elaine Scarry's study *Resisting Representation* (Scarry 1994). 'Derived from grammar – Fuller paraphrases - it simply means a word that is both a verb and a noun, a thing and a motion' (Fuller 2003: 34). What is important at this point is that, as a participial - that is, non totally static - object, software can itself produce some form of criticism. In other words, Fuller suggests that, rather than on the critique of software, we focus on alternative software production. He wishes for 'models of software production that contain engines for its theorization' (22). He identifies three models of alternative software production: critical software, social software and speculative software.

Critical software is 'software designed explicitly to pull the rug from underneath normalised understandings of software' (22). According to Fuller, critical software operates in two key modes: by using the evidence presented by normalised software to allow the conditions of software to become manifest – for instance, the installation 'A Song for Occupation' which maps the interface of Microsoft Words to 'reveal the blue-grey labyrinth in which writing is so happily lost' (23) - or instances of software 'that runs like a normal application, but has been fundamentally twisted to reveal underlying constructions (of the user, the coding, etc.)' (30). Social software is built by and for those who are 'normally locked out of its production' – that is, by and for people that do not take part in industrial software production. For Fuller free software and open source are examples of social software. Finally, speculative software has a tight relationship with fiction and art. It reinvents and expands upon existing languages, it 'explores the potentiality of all possible programming' (30). Fuller considers it a 'reinvention of software by its own means' (30). In sum, Fuller suggests that the best critical approach to software is the production of alternative software that 'twists' or 'reveals' what is normally 'behind' software itself. Fuller's image of the 'blip' is of the utmost importance here. For him, 'behind the blip' we can find the social, economical and political realm – and

speculative software wants to 'intercept', 'map' and 'reconfigure' the social, economical and political 'by means of the blips' (Fuller 2003: 31). According to Fuller, 'blips' are not signifiers 'but integral and material parts of events which manifest themselves digitally' (31). Speculative software is able to 'operate reflexively upon itself and the condition of being software' (32), and makes visible the dynamics of the social and economical events it connects to.

Once again, Fuller's argument on the 'blip' appears quite problematic. While being aware that software *is* a social object, I want to ask here whether Fuller's idea of 'making the social apparent' in software conveys a dream of transparency. In other words, exactly what kind of demystification (if any) is at work in critical and speculative software? What I want to suggest is that critical and speculative software always run the risk of substituting demystification with some other mystification - that is, the mystification of immediacy, the mystification of demystification. Once again, and in order to avoid such a dead end, a better approach would be to keep in mind that – as I have argued earlier on in this chapter - contemporary technology (including software) is already in deconstruction. But deconstruction does not aim at making apparent 'the social' behind and through software ('the blip'). It rather aims at dealing with the unforeseen consequences of technology and with its capacity of bringing forth the unexpected.

To conclude, I argue that, in order to understand the importance of software (and broadly of technology) for the constitution of the human, it is necessary to consider how the technology named 'software' emerges in specific technological and disciplinary contexts over time. Each of these emergences must be investigated in its singularity to uncover its significance for an understanding of the relationship between technology and the human - and ultimately for a political understanding of technology. In the following chapter I will show how, in the late 1960s, 'software' emerged as a specific construct in relation to 'writing' and 'code' in the discourses and practices of Software Engineering.

3 Software as Material Inscription

The Beginnings of Software Engineering

> In the beginning was the word, all right – [general laughter] but it wasn't a fixed number of bits!
>
> (Naur and Randell 1969: 50)

In an article published in 2004 in the *Annals of the History of Computing,* Michael S. Mahoney writes:

> Dating from the first international conference on the topic in October 1968, software engineering just turned thirty-five. It has all the hallmarks of an established discipline: societies (or sub-societies), journals, textbooks and curricula, even research institutes. It would seem ready to have a history. Yet, a closer look at the field raises the question of just what the subject of the history would be.
>
> (Mahoney 2004: 8)

Mahoney points out that, although it is not hard to find definitions of Software Engineering throughout technical literature – for instance, in his 1989 paper entitled 'The Software Engineering Process', Watts Humphrey, a leading practitioner in the field, defines it as 'the disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software' (Humphrey 1989b: 82) - it is also rather easy to come across doubts as to whether Software Engineering's current practice meets such criteria. For example,

in an article published at about the same time, Mary Shaw - herself a distinguished scholar and practitioner in the field of Software Engineering - muses whether Software Engineering was an engineering discipline at all, and states: 'Software engineering is not yet a true engineering discipline, but it has the potential to become one' (Shaw 1990: 15). 'From the outset', Mahoney comments, 'software engineering conferences have routinely begun with a keynote address that asks "Are we there yet?" and proposes yet another specification of just where "where" might be' (Mahoney 2004: 8). His own article, he adds, stems from such an address, delivered to ACM SIGSOFT's 9th Foundations of Software Engineering Conference (FSEC 9) in 1998.

Being interested in writing a history of Software Engineering, Mahoney is particularly troubled by the fact that 'the field has been a moving target for its own practitioners' from its very beginning, and that practitioners openly disagree on what it is. Historians – he suggests – can as readily write a history of Software Engineering 'as the continuing effort of various groups of people engaged in the production of software to establish their practice as an engineering discipline' - that is, a history of this process of self-definition (8). Such an approach would immediately pose a number of questions, such as which model of engineering software practitioners refer to; moreover, it would place the history of software engineering within what Mahoney terms 'the comparative context of the history of professionalization and the formation of new disciplines' (8). For instance, an oft-quoted passage from the introduction to the proceedings of the first NATO Conference on Software Engineering, held in Garmisch in 1968, declares: '[t]he phrase "software engineering" was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering' (Naur and Randell 1969: 13).

As Mahoney notices, the definition of Software Engineering given in the proceedings of the Garmisch Conference is provocative indeed, since it leaves all the crucial terms undefined. For instance, it is unclear what 'manufacturing' software means, or what the 'theoretical foundations and practical disciplines' that underpin the 'established branches of engineering' are. Furthermore, the term

'traditional' hints at a search for historical precedents, or what is commonly referred to as 'roots' (Mahoney 2004: 8) – and yet the above passage seems to leave open the question of how the existing branches of engineering have taken their present form. On the other hand, the definition of Software Engineering also concerns its 'agenda' - that is, what the practitioners of the field agree ought to be done, 'a consensus concerning the problems of the field, their order of importance or priority, the means of solving them (the tools of the trade), and perhaps most importantly, what constitutes a solution' (8). As I will detail later on in this chapter, much of the disagreement among the participants in the first NATO Conference on Software Engineering rested both on their different professional backgrounds and on the conflicting agendas they brought to the gathering. None of them – as Mahoney emphasizes - was a software engineer, 'for the field did not exist' (9). Rather, they came from quite varied professional and disciplinary traditions.

Albeit Mahoney points out that Software Engineering has always been characterized by strong self-reflexivity, this does not lead him to question the concept of disciplinarity. He rather presents self-reflexivity as an inconvenience – something that needs to be fixed. And yet, as I explained in Chapter Two, the fact of being a 'moving target' makes Software Engineering particularly valuable as a starting point for my investigation of software. In Chapter Two I suggested that software's singularity as a technology resides precisely in the relationship it entertains with writing, or, to be more precise, with a historically specific form of writing. In this chapter I want to illustrate the co-emergence of software and writing - or, even better, of 'software', 'writing' and 'code' as constantly shifting terms which are actually constitutive of one another - in the discourses and practices of Software Engineering in the late 1960s, at the beginnings of the discipline.[1] I am not interested in producing an historical overview of the field in Mahoney's sense. Instead, I want to investigate the way in which Software Engineering has been instituted as a discipline – a process that involved the establishment of its own

[1] In Chapter Two I argued for a 'singular' understanding of software inspired by Gary Hall's understanding of the term 'singularity'. Following Hall (2007a), I argued that it is not enough to take into account the 'specificity' of software as a technology. Attention must also be paid to the 'singular' ways in which software is understood and operates in various historical moments and contexts. For this reason, as I also remarked in Chapter Two, although the co-emergence of 'software', 'writing' and 'code' constitutes the historical specificity of Software Engineering, it cannot be said that it also constitutes the specificity of 'all software'.

object ('software'), itself involved in a mutually constitutive relationship with two other entities ('writing' and 'code'). In order to do this, in this chapter I present a close reading of the foundational texts of Software Engineering – namely, the reports of the first and second Conferences on Software Engineering, convened by the NATO Science Committee respectively in 1968 in Garmisch (Germany) and in 1969 in Rome (Italy) (Naur and Randell 1969; Buxton and Randell 1970).[2]

My analysis is based mainly on the report of the Garmisch conference - the first ever conference on Software Engineering. (One year later, things had changed significantly in Software Engineering and the climate of the conference that took place in 1969 in Rome was very different and far less enthusiastic than that of the first one. The main point of interest in the Rome report is a growing awareness of the lack of communication between software practitioners and the academic world – something that did not seem to affect the participants of the Garmisch conference. However, all the essential issues of Software Engineering are set out in the Garmisch conference report.) In this chapter I also examine a number of later texts that comment upon the NATO conferences and provide additional information about them (such as Galler 1989; Gries 1989; Randell 1979, 1998; Shaw 1989, 1990).

The first NATO Conference on Software Engineering was conceived by the Study Group on Computer Science, established in the autumn of 1967 within the NATO Science Committee (Naur and Randell 1969: 13). As Brian Randell – editor of the reports of the 1968 and 1969 conferences – recalls later on in his article 'Software Engineering in 1968', it was 'the Garmisch conference that started the software

[2] The report of the first NATO Conference on Software Engineering, held in Garmisch from 7th to 11st October 1968, was edited by Peter Naur and Brian Randell soon after the conference. NATO was in charge of the actual printing and distribution, and the report became available three months after the conference, in January 1969 (Naur and Randell 1969). The report of the second conference, held in Rome from 27th to 31st October 1969, was edited by John Buxton and Brian Randell and published in April 1970 (Buxton and Randell 1970). Both reports were later republished in book form (Buxton, Naur and Randell 1976). In 2001 Robert M. McClure made both reports available for download in pdf format at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/. The pagination of the pdf version slightly differs from the original printed version. All the references made in this chapter are based on the original pagination.

engineering bandwagon rolling' (Randell 1979: 1).[3] Although, as I have shown above, the term 'Software Engineering' still retained much of its novelty and its provocative potential in 1968, the need for such a discipline had already been discussed by Wallace J. Eckert at the 1965 Fall Joint Computer Conference (Gordon 1968: 200; Randell 1979: 2). However, according to Randell, one of the most significant aspects of the Garmisch conference was the willingness of the participants to admit 'the extent and seriousness of current software problems' (1) at a time when technical literature tended to sound rather celebratory. To give but one example, in the preface to the collection of essays *Advances in Computers*, which he edited in 1968 with Franz L. Alt, Morris Rubinoff wrote enthusiastically:

> There is a seductive fascination in software. There is a pride of accomplishment in getting a program to run on a digital computer, a feeling of the mastery of man over machine. And there is a wealth of human pleasure in such mental exercises as the manipulation of symbols, the invention of new languages for new fields of problem solving, the derivation of new algorithms and techniques, the allocation of memory between core and disk - mental exercises that bear a strong resemblance to such popular pastimes as fitting together the pieces of a jigsaw puzzle, filling out a crossword puzzle, or solving a mathematical brainteaser of widely familiar variety. And what is more, digital computer programming is an individual skill, and one which is relatively easy to learn.
>
> (Alt and Rubinoff 1968: 9).

Although Randell does not perform such an analysis, I want to highlight here the rhetoric of instrumentality at work in Alt and Rubinoff's passage: they dwell upon

[3] The Garmisch conference report describes the participants as 'about 50 experts from all areas concerned with software problems - computer manufacturers, universities, software houses, computer users, etc.' (Naur and Randell 1969: 13). The fact that the participants had quite heterogeneous backgrounds was in line with the intentions of the organizing committee (Randell 1979). Nearly half of them came from North America, the rest from various European countries, and among them there were outstanding scholars and practitioners, many of whom were meeting each other for the first time. Many went on to write fundamental works in Software Engineering in the following decades. A list of participants is given in Naur and Randell (1969: 212-217).

the mastery of 'man' over technology, and they associate it, as Alfred Gell would have it, with the 'enchantment' of technology (or the 'seductive fascination' of software).[4] Even more importantly, 'symbols' make their appearance here as something that can be manipulated - that is, as tools - in order to produce the magic wonders of technology.

Randell contrasts Alt and Rubinoff's passage with Edsger W. Dijkstra's declaration at the Garmisch conference that '[t]he general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement' (Naur and Randell 1969: 121). I will return to Dijkstra's complex passage later on in this chapter. For now, suffice it to say that, as Randell also points out, terms such as 'software crisis' and 'software failure' were largely used during the Garmisch conference, and that for this reason many of the participants viewed the conference as a turning point in their way of approaching software. In the following section of this chapter I will consider Randell's remark in more depth, and argue that, with the Garmisch conference, software indeed began to be conceptualized *as a problem*. Moreover, the so-called 'software crisis' was constituted as the point of origin for the discipline of Software Engineering.

Indeed, since its very inception, the Garmisch conference report defines software as a problem. The editors clearly state that the conference and the report both deal with 'a problem crucial to the use of computers, viz. the so-called software, or programs, developed to control their action' (Naur and Randell 1969: 3). Not only is software here conceptualized as a problem – it is also presented as crucial to the deployment of computers, since it is responsible for controlling them. Thus, from the very first lines of the report, the relationship between software and control is established, while at the same time this relationship is characterized as problematic.

[4] As I pointed out in the Introduction, Alfred Gell (1992) argues that the foundation of art is a technical level of excellence that a society misrepresents to itself as a product of magic. Gell understands art as a special form of technology and art objects as the means for obtaining social consensus. This is why he claims that 'the power of art objects stems from the technical processes they objectively embody: the *technology of enchantment* is founded on the *enchantment of technology*' (Gell 1992: 44). Similarly, technology is capable of casting a spell over us so that we see the world in an enchanted form. In other words, the magical prowess that is supposed to have entered the making of the art object or of the technological artefact relies on the level of cultural understanding that surrounds it.

Before examining in detail the way in which software is problematized in the Garmisch conference report, it is worth spending a little time on the structure of the report itself, in order to facilitate its reading. In his article of 1996 Randall recalls that it was the participant Fritz Bauer who emphasized the importance of providing a report of the conference and who persuaded Randell himself and Peter Naur to be its editors (Randell 1998: 51).[5] In the preface to the report, the editors give a minute explanation of the way in which the report itself was put together. They write:

> The actual work on the report was a joint undertaking by several people. The large amounts of typing and other office chores, both during the conference and for a period thereafter, were done by Miss Doris Angemeyer, Miss Enid Austin, Miss Petra Dandler, Mrs Dagmar Hanisch and Miss Erika Stief. During the conference notes were taken by Larry Flanigan, Ian Hugo and Manfred Paul. Ian Hugo also operated the tape recorder. The reviewing and sorting of the passages from written contributions and the discussions was done by Larry Flanigan, Bernard Galler, David Gries, Ian Hugo, Peter Naur, Brian Randell and Gerd Sapper. The final write-up was done by Peter Naur and Brian Randell. The preparation of the final typed copy of the report was done by Miss Kirsten Anderson at Regnecentralen, Copenhagen, under the direction of Peter Naur.
>
> (Naur and Randell 1969: 11 f.)

In fact, after the Garmisch conference, Randell and Naur spent an extra week editing the draft report, which was provided jointly by the people listed in the first

[5] A distinguished academic, Fritz Bauer was among the developers of Algol (short for ALGOrithmic Language), one of the first programing languages, in the mid 1950s. Randell had met him before the Garmisch conference at the meetings of the IFIP Algol Committee in Vienna. Bauer was affectionately nicknamed 'Onkel Fritz' by the other members of the Committee. In fact, Randell attributes the idea for the first NATO Conference on Software Engineering (as well as for the use of the term 'Software Engineering' as the conference title) to Bauer.

section of the above passage.[6] The first problem that had to be addressed concerned the structure of the report. Randell and Naur had to choose between two basic forms of classification: the one based on the sequence of 'steps' involved in the process of software development (from the project's initial phase, through design, production, distribution and maintenance), the other based on different 'aspects' of software development (such as documentation, management, communication, programming techniques and hardware considerations) (Naur and Randell 1969: 10). As it will become clearer throughout this chapter this choice was not trivial, since the identification of both the 'steps' and the 'aspects' of software development were pivotal and highly controversial topics of the conference itself. Furthermore, the two competing models for the structure of the report were also discussed during the conference, and the participants were very aware of the importance of their decision. Their awareness, as Randell recalls, was due to the fact that 'a tremendously excited and enthusiastic atmosphere developed at the conference', motivated by the participants' growing appreciation of the shared concern about the 'software crisis'. Therefore

> general agreement arose about the importance of trying to convince not just other colleagues, but also policy makers at all levels, of the seriousness of the problems that were being discussed. Thus throughout the conference there was a continued emphasis on how the conference could best be reported. Indeed, by the end of the conference Peter [Naur] and I had been provided with a detailed proposed structure for the main part of the report. This was based on a logical structuring of the topics covered, rather than closely patterned on the actual way in which the conference's various parallel and plenary sessions had happened to be timetabled. Peter and I were very pleased to have such guidance on the structuring and general contents of the report, since we both wished to create something that was truly a conference report, rather than a mere personal report on a conference that we happened to have attended.
>
> (Randell 1998: 51)

[6] It is quite obvious from the above quotation that the working group was gendered in a particular way, with the women doing all the secretarial tasks and the male participants in the conference being in charge of proper editing. The different media involved in the production of the report were also controlled by men, except for typing. The report of the second NATO Conference on Software Engineering shows that things were not much different in Rome one year later (Buxton and Randell 1970).

This long passage shows how the structure of the report is justified by its editors by means of the criteria of legibility, as well as by positioning it as a tribute to the participants' joint effort to produce an effective record of the conference. However, the most important point here is Randell's focus on 'the structuring and general contents of the report'. Looking back to the Garmisch conference in 1996, he seems quite conscious that the structuring of that report has had a lasting effect on the structure of Software Engineering as a field.

Actually, as the editors make clear in the preface to the report, the final structure is based on the first kind of classification - that is, it follows the step-by-step model of the process of software development. Nonetheless, in many passages of the report the second type of classification 'creeps in' – and the editors attempted to mitigate its effects by the provision of 'a detailed index' (Naur and Randell 1969: 10).[7] Now it must be emphasized that such a structure based on the successive 'steps' or 'phases' of software development has later become a universal model for the production of manuals and reference texts of Software Engineering. Is that 'and' in Randell's expression 'the structuring *and* general contents of the report' - one might wonder - really an 'and'? Could it be said that 'structure' and 'content' are not so clearly divided, especially at the moment of the constitution of a discipline? As I will explain later on in this chapter, the way of writing 'about' Software Engineering actually enacted and produced 'Software Engineering' not only as a discipline but also as a set of techniques or instructions for writing software, including computer programs. In this sense, writing seems to be constitutive of both the discipline of Software Engineering and of what Katherine Hayles would name 'code'. Indeed, writing is not the opposite of 'code', or a slightly impoverished version of 'code', as Hayles (2005) seems to suggest. In order to develop this point, let me look briefly at the way in which the report was materially produced.

[7] Indeed, the conference had been planned in quite an unconventional way: rather than taking part in a series of panels at which papers were presented and discussed, the participants were divided into three groups corresponding to the topics of 'Design', 'Production' and 'Service', and engaged in parallel discussion-based working sessions punctuated by the occasional plenary session (Naur and Randell 1969: 13; Randell 1979: 6). Since design, production and service were the three main parts in which the process of software development could be roughly divided at the time, the original structure of the conference – as it was intended by the organizing NATO committee – was based on the classification by 'steps'.

As the editors explain, the report is not merely a collection of the working papers contributed by the participants during or before the conference. Rather, it results from the reworking of such papers *and* of the discussions that took place at the conference.

> The discussions were recorded by several reporters and most were also recorded on magnetic tape. The reporters' notes were then collated, correlated with footage numbers on the magnetic tape, and typed. Owing to the high quality of the reporters' notes it was then, in general, possible to avoid extensive amounts of tape transcription, except where the accuracy of quotations required verification.
>
> (Naur and Randell 1969: 10)

First of all, the above passage clarifies that, in order to document the discussions, different media were involved. The discussion was taped and then selectively transcribed - therefore parts of it were lost in the process, since the correlation between the tape and the written notes functioned as a kind of filter. What I want to emphasize here is that the report constitutes a narrative based on what, from an ethnographic point of view, could be considered fieldwork notes, plus a certain amount of magnetic tape not entirely transcribed. The editors justify the inclusion of 'many direct quotations and exchanges of opinion' in this narrative with their intent of having the report 'reflect the lively controversies of the original discussion' (3). To paraphrase Naur and Randell, it might be said that they view the inclusion of different quotations as the incorporation of conflicting point of views. They even give an interesting example of their work in the preface of the report.

> ... here is an example, albeit extreme, of the typed notes:
>
> 536 DIJKSTRA
>
> F -
>
> H --
>
> P --?--
>
> (here '536' is the tape footage number, and the letters F, H and P identify the reporters). This section of the tape was transcribed to reveal that what was actually said was:

> "There is a tremendous difference if maintenance means adaptation to a changing problem, or just correcting blunders. It was the first kind of maintenance I was talking about. You may be right in blaming users for asking for blue-sky equipment, but if the manufacturing community offers this with a serious face, then I can only say that the whole business is based on one big fraud. [Laughter and applause]".
>
> (Naur and Randell 1969: 10 f.)

The example that the editors so proudly provide shows how a kind of shorthand comment results in an elaborated sentence which also includes a joke – and a cheerful reaction on the part of the audience – in the final report (with the mediation of the magnetic tape). It is quite clear at this point that the production of the report's narrative in the week following the conference involved various processes of selection, sorting out and summing up of numerous texts. Moreover, in the report many quotations were sparsely interpolated by the editors' comments. Randell recounts that he and Naur argued that they should not provide any additional text as editors, but rather build the core of the report 'merely by populating the agreed structure with suitable direct quotations from spoken and written conference contributions'.[8] Nevertheless, it was Randell's opinion that 'brief editorial introductions and linking passages' would improve the overall legibility and consistency of the text – and this eventually became the final form of the report. A short selection of the working papers contributed by participants was also incorporated in full as appendices (Randell 1998: 52).

What I want to emphasize here, on the basis of the quotations presented above, is that the narrative of the report has fundamentally shaped the field of Software Engineering through the selection, inclusion and exclusion of problems and topics. In Mahoney's words, it could be said that the Garmisch conference report constitutes the 'agenda' of the field (Mahoney 2004: 8). In Gary Hall's terms, it could be said that the report is the first attempt at building an 'archive' for Software

[8] Later on, in 1979 Randell restated that 'in an attempt to avoid undue editorial bias, [he and Naur] evolved an editorial style which relied heavily on the use of direct quotations to flesh out the structure that had been agreed by the meeting' (Randell 1979: 6).

Engineering and at 'performing' Software Engineering as a discipline (Hall, 2007a). To understand this point better, let me follow Hall's argument for a little while.

In a his 2007 paper presented at the 'Remediating Literature' Conference at Utrecht University, Gary Hall points out that every archive regulates what can be collected, stored and preserved. He writes:

> It is important to realise that an archive is not a neutral institution but part of specific intellectual, cultural, technical and financial networks. An archive's medium, in particular - be it paper, celluloid or tape - is often perceived as constituting merely a disinterested carrier for the archived material. Yet the medium of an archive actually helps to determine and shape its content; a content, moreover, which is performed differently each time, in each particular context in which it is accessed and material is retrieved from the archive.
>
> (Gary Hall 2007a: non-pag.)

In this paper Hall focuses on CSeARCH, a digital archive of works in cultural studies which, being based on open access technology, not only enables scholars to publish and archive research literature in a much more flexible way than traditional forms of publication, but also changes the definition of what research literature is. For instance CSeARCH allows the archivization of drafts, leaflets, posters, personal correspondence, multimedia resources and, as Hall puts it, 'laundry notes and scraps' like the one stating '"I have forgotten my umbrella", which was found among Nietzsche's papers after his death and about which Derrida has written at length'.[9] It is the specific structure of the open access digital archive that shapes what it preserves and classifies as legitimate scholarship, in both time and scope. By doing so, the digital (open) archive changes what is considered legitimate cultural studies – that is, it changes the definition of the discipline. In this sense, whenever accessed and consulted, the archive constitutes an instance of the discipline; the very act of accessing the archive of selecting, downloading or uploading texts 'performs' a singular instance of the discipline of cultural studies. Hall explains:

[9] Hall refers here to Derrida (1979: 139).

> Consequently, a digital cultural studies archive is not just a means of reproducing and confirming existing conceptions of cultural studies; of what cultural studies already is or is perceived as having been. It is *partly* that. But it's also a means of *producing* and *performing* cultural studies: both what it's going to look like in the past; and what there's a chance for cultural studies to have been in the future.
>
> (Gary Hall 2007a: non-pag.)

As I have pointed out earlier on in this chapter, Hall is very attentive to the singularity of new technologies and to the singular relationship that CSeARCH entertains with the discipline of cultural studies. Here I want to argue that the same attention must be paid to the material production of the Garmisch conference report and to its singular relationship with Software Engineering. The Garmisch conference report was based on media technology such as the magnetic tape recorder and the typewriter, and although it included different points of view on the topic of Software Engineering, yet it framed these points of view within a narrative controlled by the editors, thus giving them a certain degree of fixity. Furthermore, after it was printed and circulated, the report could not be changed any further and therefore it contributed to the stabilization of Software Engineering as a disciplinary field, at least to some extent. In Hall's terms, the Garmisch conference report 'performed' the first version of the field of Software Engineering.

The significance of the Garmisch conference report for shaping Software Engineering is vastly documented. For instance, the contents list of Software Engineering textbooks from the late 1990s has not changed much from that of the Garmisch conference report.[10] Twenty years after the NATO conferences, in a short article published in 1989, Mary Shaw, who was a graduate student at Carnegie Mellon at the time of the first conference, comments:

> The thing that fascinates me most about the Garmisch Proceedings is how fresh they are even now, twenty years later. The introductory

[10] In Chapter Four I will examine a number of Software Engineering reference texts from the 1980s and 1990s in order to expand on this point.

> highlights [Naur and Randell 1969: 3] list the major topics of discussion ... Of these topics, only the last [about the separate pricing of hardware and software] has been settled. The remainder form a pretty good example of the usual problem list for software engineering. ... Not only are the topics still the same, but many of the problems still sound fresh.
>
> (Shaw 1989: 100)[11]

Even more important than the persistence of the central topics of Software Engineering is the 'freshness' of the open problems that Shaw mentions. Indeed, many of these problems were intentionally left open in the Garmisch conference report, as the editors notice:

> In order to retain the spirit and liveliness of the conference, every attempt has been made to reproduce the points made during the discussion by using the original wording. This means that points of major disagreement have been left wide open, and that no attempt has been made to arrive at a consensus or majority view. This is also the reason why the names of participants have been given throughout the report.
>
> (Naur and Randell 1969: 11)

We are brought back here to Mahoney's remark that the field of Software Engineering has been a 'shifting target' for its own practitioners for decades and that it still is (Mahoney 2004: 8). It can be said that the problems that were left open in Garmish have remained essential to the field *as* open problems and as the subjects of an ongoing discussion. One might also wonder at this point as to what the participants' reactions were after they had been given the final report. Did anyone ever claim that 'he did not say that', or at least not 'in that way'? Randell gingerly recounts: '[m]y memory tells me that [the] draft was then circulated to participants for comments and corrections before being printed, but no mention is made of this in the report so I may be wrong' (Randell 1998: 52). Apparently, then, the final feedback from the participants (if there was any) did not lead to any

[11] Mary Shaw's note, 'Remembrances of a Graduate Student', was presented at the panel 'A Twenty Year Retrospective of the NATO Software Engineering Conferences' of the 11th International Conference on Software Engineering, and published in the Conference Proceedings (Shaw 1989).

significant modification of the report. As Randell himself relates, Doug McIlroy, one of the participants, famously described the report as 'a triumph of misapplied quotation' (Randell 1998: 52). However, the participants must have been conscious that the report represented a turning point in their professional and scholarly practice, because significantly Mary Shaw recalls that Al Perlis, who at the time taught at the Carnegie Mellon, gave her and the other graduate students copies of the report with the words, 'Here, read this. It will change your life' (Shaw 1989: 99).

To summarize the above argument, in the Garmisch report Software Engineering establishes its own narrative as a discipline by setting itself a starting point (the 'software crisis') and a past (the tradition of engineering), as well as by opening up a future (its agenda). The report constitutes a re-staging of the discussion that took place in Garmisch, which is re-mediated - Bolter and Grusin's term (2002) sounds apt here - between handwritten notes, magnetic tape and selective typewriting in order to reach the final form of the report. Given the selective transcription of the tape, it can also be said that a part of the discussion has been (irremediably) lost in re-mediation.

Before going on to investigate in what way the Garmisch conference changed the professional practices of software developers in the late 1960s, let me give an example of the external appearance of the text of the report. A brief examination of the two pages of the report included in Appendix A (Naur and Randell 1969: 15-18; http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF: 9f.) shows that the Garmisch report is articulated as a sequence of quotations, either taken from transcribed debates or from working papers (in which case the title of the paper is given in parentheses next to the author's name). These quotations are interpolated throughout the report by the editors' comments (italicized), which do not just express opinions, but in fact structure a narrative by integrating the quotations themselves into a discourse, by making explicit their logical and/or temporal connections, and sometimes even by giving information about the general reactions of the audience. For instance, p.9 of the report starts with an italicized introduction from the editors that constitutes both an abstract of the following section and a contextualization of the topic, followed by a sentence (again by the editors) that

introduces three quotations from the transcribed material ('First, three quotations which indicate the rate of growth of software'). After this the three quotations are listed by author, almost as different voices in a script. An interjection from the editors - starting with 'Yet...' – follows and introduces a corrective statement from another participant. A further comment from the editors corrects the above, followed and supported by two other quotations. So far, the facts (the dramatic rate of growth of software) and different opinions about them (the alarm raised by this growth, as well as some praise for the successes in the field) have been presented. Then the editors offer another comment, starting with 'However...', which returns to the concern expressed by the majority of the participants regarding the contemporary rate of the growth of software. This comment is supported by three other quotations. At this point, the editors suggest the need for a discipline named 'Software Engineering' – without defining it, but implying a large consensus among the conference participants on the necessity of developing such an approach. Five quotations are provided which diffusely lament the immature status of the current techniques of software production; they are followed by a more optimistic statement reinforced by a quotation. After this, the editors offer another comment that leaves the problem of the immaturity of (and of the need for) Software Engineering substantially open, and they back it up with two more quotations. Finally, the editors conclude: 'This being the case, perhaps the best quotation to use to end this short section of the report is the following', thus making explicit their uncertainty in the selection of a conclusion and remarking on the openness of the question. The final comment is followed by just one quote that in fact leaves the question open.

In the structure that I have just described, the editors' comments act as connectors between quotations, giving the report an argumentative structure while framing the conference debates. Although the report claims to leave many problems open, one might argue that there is a certain degree of closure in the superimposition of an argumentative structure on the different points of view that emerged during the conference. At first glance, the text of the report actually seems to be articulated on two different levels: the level of temporal duration (since it restages the conference debate) and the level of logical structure (since it also extrapolates quotations from their original context and uses them to 'flesh out' an argumentative skeleton). Moreover, the voice of the editors sounds like a unitary, impersonal voice that, by

framing the quotations, enters into a dialogue with them, while also exercising a kind of containment on the different voices emerging from the conference and placing them in a particular structure – namely, the structure of the disciplinary field of Software Engineering. In a way, it could be argued that the editors' voice must have a unitary tone precisely in order to provide a consistent frame for the divergent points of view of the conference participants. However, I want to suggest that it would be futile to ask whether such an enclosure deprives the quotations of their original potential as arguments. In fact, as I argued in Chapter Two, following Jacques Derrida, 'citationality' – or 'iterability' - is a general characteristic of all language. Indeed, language would not be able to function *as* language if it could not be quoted. 'What would a mark be that could not be cited?', asks Derrida in *Limited Inc.* (Derrida 1988: 12), and adds: 'a written sign carries with it a force that breaks with its context, that is, with the collectivity of presences organizing the moment of its inscription' (9). Therefore, one of the characteristics of writing is that it can continue to 'act' as writing - that is, to be readable - 'even when what is called the author of the writing no longer answers for what he has written, for what he seems to have signed' (8). Indeed, every single quotation in the report works in the context of the italicized argument (which is not the context of the conference, nor the context of the impression left by the participants' voices on magnetic tape) precisely in order to 'perform' the new field of Software Engineering.[12]

Also in order to establish Software Engineering as a discipline, the Garmisch conference report mentions its readers, who are also the potential beneficiaries of such an emerging new field of knowledge. Let me now examine how this process works in the report by attempting to ascertain, first of all, who the report is intended for. To answer this question it is worth taking a look at the context in which the conference was organized – that is, at what Randell calls 'the 1968 software scene' (Randell 1979: 2). The problems that the Garmisch conference wanted to deal with were mainly related to 'large' or 'very large' software-systems – that is, systems of

[12] It is worth noting that the Rome conference report follows the format of the Garmisch one. Randell (1998) recalls that the second conference was 'far less harmonious and successful than the first', and that it bore little resemblance to it. Nevertheless, in the absence of a clear brief from the participants in the Rome conference regarding the structure and content of the report, the editors decided to reproduce the structure of the Garmisch report, while declaring in the Preface that the similarities between the reports were purely 'superficial'. Thus, the Rome report embodies a much more stabilized version of Software Engineering.

a certain complexity whose development required a conspicuous effort in terms of time, money and the number of professionals involved. In the late 1960s the unit of measure for determining whether a system was 'large' was the number of lines of code it contained. A large software system could include several thousand lines of code. Another popular unit (still used nowadays) was the 'man-year' - that is, the number of years an average programmer would spend on a software system if they were to develop the system by themselves. A few years after the Garmisch conference, a sub-unit of the man-year – the man-month - became the title of a classic of Software Engineering, Frederick Brooks' *The Mythical Man-Month* (1995), which I will examine in Chapter Four.[13]

However, in the late 1960s quite a lot of complex software systems were being developed. As Randell states (thus explaining NATO's interest in Software Engineering), 'it was the US military-industrial complex that first started to try and develop very large software systems involving man-millennia of effort' (Randell 1979: 5). Randell mentions a paper presented by Joseph C. R. Licklider in 1969 as a contribution to the public debate around the Anti-Ballistic Missile (ABM) System (a complex project which contemplated the development of enormously sophisticated software) and eloquently titled 'Understimates and Overexpectations'. In his paper Licklider provides a vivid picture of the gap between the military's goals and their achievements. He states: '[a]t one time, at least two or three dozens complex electronic systems for command, control and/or intelligence operations were being planned or developed by the military. Most were never completed. None was completed on time or within the budget' (Licklider 1969: 118). This passage illustrates well the 'software crisis' which, as I have pointed out earlier in this chapter, is established as the founding event of the disciplinary narrative of

[13] In Chapter Four I will also investigate the temporality of what I want to name 'man-time', of which the 'man-month', the 'man-year' and even the 'man-millennium' are specific instances. As it is quite clear by now, these units of measure convey the largeness of a software system by quantifying the (estimated) duration of its development. It is important to notice that the line of code (as a measure of software, or 'software metric') is also related to time. The line of code is defined as a line in the text of a program's 'source code' – which in turn is any sequence of statements written in a computer programming language. Before being executed, the source code is converted from this human-readable format into a computer-executable one by means of some software device such as an interpreter or a compiler. In this process, the line of code is transformed into a sequence of 'tokens', which in turn, when executed, will effect internal changes in the computer in a certain temporal order. Therefore, code is related to time in a complex way, a point to which I will return in detail in Chapter Five.

Software Engineering. Even more importantly, in his recollections about Software Engineering in 1968, Randell adds the following comment:

> I still remember the ABM debate vividly, and my horror and incredulity that some computer people really believed that one could depend on massively complex hardware and software systems to detonate one or more H-bombs at exactly the right time and place over New York City to destroy just the incoming missiles, rather than the city or its inhabitants.
>
> (Randell 1979: 5).

Here Randell's horror at the excessive self-confidence of some software professionals stems from the connotative association between technology, catastrophe and death in a cold-war scenario. As we shall see in a moment, 'horror' – a powerful emotion - is the result of the anticipation of the consequences of technology combined with the awareness of its intrinsic fallibility. For now, it is worth noting that Randell's passage hints at what is in fact the most influential component of the Garmisch report's readership: the military.

However, by the late 1960s large-scale systems were not unique to the military scene. For instance, computer manufacturers had started to develop operating systems - that is, software systems that provided all the basic functionalities for the management of computers' internal resources, and that at the time came free with the hardware from the manufacturer.[14] The late 1960s operating systems were much more complicated than their predecessors – for instance, release 16 of OS/360 was announced in July 1968 and contained almost one million instructions (Belady and

[14] A famous example of those earlier operating systems was OS/360, which ran on IBM computers of the 360 series. A vastly innovative operating system, OS/360 was extremely complex for the time and, albeit becoming rapidly popular, it contained a number of technical flaws. Its development during the first half of the 1960s was characterized by many setbacks, mainly due to the poor management of time, and it was largely discussed during the Garmisch conference. Frederick Brooks became the project manager of OS/360 at IBM in 1964. In order to speed up the process of software development, he mistakenly added more programmers to the project, falling behind schedule as a result. Later on, drawing on this experience, he formulated the principle that 'adding more manpower to a late software project makes it later', which became known as 'The Brooks' Law' (Brooks 1995). I will return to Brooks' contribution to Software Engineering in Chapter Four.

Lehman 1976).[15] Specialized real-time systems were also being developed, such as the first large-scale airline reservation system, the American Airlines SABRE system.[16] Generally, these systems started out rather disastrously, but, after some time and with a lot of effort, they were reasonably improved and reached a tolerable level of performance. For instance, when it was first introduced, the SABRE system caused chaos at the airports (and a lot of extra business for competing airlines), but by 1968 it was performing quite reliably, serving around 2,000 terminals and processing almost 3,000 messages per minute at peak load times (Hopkins 1968). Operating systems and real-time systems were intensely discussed at the Garmisch conference, since the costs incurred in developing them were immense and they were very much in the public's eye. Moreover, some of these systems (such as TSS/360, an alternative to the operating system OS/360) kept performing poorly notwithstanding the vast amount of resources lavished on them by their manufacturers – and the professionals involved in these projects felt the pressure of the public opinion.

It should be quite clear by now why the editors of the Garmisch conference report emphasize that, while being specifically addressed to 'the immediate users of computers and to computers manufacturers', the report may also 'serve to enlighten and warn policy makers at all levels' (Naur and Randell 1969: 3). Indeed, they expressly state that every effort was undertaken to make the report 'useful to a wide circle of readers'. Specific parts of it, the editors declare, 'are written for those who have no special interest in computers and their software as such, but who are concerned with the impact of these tools on other parts of society' (9). Potential readers are 'civil servants, politicians, policy makers of public and private enterprises' (9). According to the editors, a somewhat narrower readership encompasses all those who do not work in the software field but nevertheless need

[15] A release is a 'stable' state of a software system – namely, a version of the system that is deemed complete and correct enough to be delivered to its users. Such delivery does not prevent system developers from improving and extending the system until they reach a new (and supposedly better) stable version of it. Releases are usually indicated by consecutive numbers. An instruction is usually a single line of code containing a basic command or data. The sequencing of instructions constitutes a program. The lines of code (or instructions) were also used as a measure of the size of software systems. Release 16 was a considerably large system for the time. In fact, at the 1968 conference E.E. David pointed out that OS/360 had by then absorbed 5,000 man-years of work (Naur and Randell 1969: 15).

[16] A 'real-time' system is a software system which is able to respond to changes in its environment 'as soon as they happen', that is in a fast and effective way. As I explained in Chapter One, real-time systems tend to be complex, high-risk, low fault-tolerance systems.

an understanding of the nature of the new field of Software Engineering; these would be 'managers of business enterprises using computers, researchers in fields other than software engineering and computer science, university officials, computer marketing personnel' (9). Finally, the report is intended for those directly engaged in software development. Painstakingly, in the preface the editors provide a list that indicates which sections of the report would be interesting for which separate group of readers. For instance, the widest audience is directed to Section 1 (Background of Conference) and Section 2 (Software Engineering and Society). Importantly, while gradually narrowing down its focus from general to technical audiences, the list never mentions the military. But, as Randell remarks in a later study, although the military first started developing large software systems in the United States, 'the secrecy that shrouded their purposes served also to hide the extent to which such projects were characterised by "underestimates and overexpectations"' (Randell 1979: 5). After all, the Garmisch conference was organized by NATO, and indeed the report sounds like an argument to persuade the NATO Science Committee that their money was well spent and that possibly more conferences on Software Engineering should be funded. The important point here is that the constitution of Software Engineering as a discipline is performed in the Garmisch report also as a form of persuasion. The military is the silent, invisible addressee of the rhetoric of the report – it is presupposed as the witness of the constituting act of the discipline performed by the report. But in what way is the relationship between Software Engineering and the 'rest of society' configured in the report?

As Naur and Randell clearly state in the preface, the readers are supposed to read the report instrumentally – that is, to use it as a tool to anticipate and evaluate the consequences of technology in time. The points of interest for readers other than software professionals are highlighted as:

- the problems of achieving sufficient reliability in the data systems which are becoming increasingly integrated into the central activities of modern society
- the difficulties of meeting schedules and specifications on large software projects

- the education of software (or data systems) engineers
- the highly controversial question of whether software should be priced separately from hardware.

(Naur and Randell 1969: 3)

Apart from the problem of education, which clearly hints at the academic ambitions of Software Engineering from its very inception, the above points clearly show that society at large is viewed as mainly concerned with the problem of the reliability of software and with its costs. Moreover, the relation between software and society (understood in terms of 'impact') is reflected in the very structure of the report. Since 'one of the major motivations for the organizing of the conference was an awareness of the rapidly increasing importance of computer software systems in many activities of society' (9), the editors abstract the representative discussions of questions regarding the 'impact of software engineering on society' from later sections of the report and collect them in an introductory section, conveniently titled 'Software Engineering and Society'. Let me now examine this section briefly in order to illustrate how in the report society is constituted as an entity separate from technology and how their relationship is defined.

As I have explained earlier on in this chapter, the report describes software growth (intended both as the increasing complexity of software and the increasing reliance of society on software systems) as the pre-eminent motivation for the Garmisch conference. Hal Helms is reported to have presented these dramatic figures at the Garmisch conference: 10,000 installed computers in Europe alone, a number increasing 'at a rate of anywhere from 25 per cent to 50 per cent per year'; furthermore, software development would soon involve 'more than a quarter of a million analysts and programmers' (Naur and Randell 1969: 15). The situation is not only measured in terms of the number of software professionals involved, but also of cost and effort. As I have mentioned above, during the conference E. E. David pointed out that T.J. Watson - IBM's founder - had estimated the cost of OS/360 development at over $50 million dollars a year, and at least 5000 men-years, while TSS/360 was probably in the 100 man-year category. The speed of software growth, according to the editors, was perceived by the conference participants with more 'alarm than pride' (15).

One must be reminded at this point of the importance of the relationship between contemporary technology and time. In Chapter One I noticed how, according to Bernard Stiegler, contemporary technology has a totally new relation with time, which becomes apparent both through the speed of technical change and the ruptures in event-ization that this change provokes. In Stiegler's words, 'there is today a conjunction between the question of technics and the question of time' that 'calls for a new consideration of technicity' (17). However, Stiegler hints to the 'dis-adjustment' between society and technology due to the speed of the latter. In a sense, the Garmisch report is concerned precisely with this problem. And yet, as we shall see in a moment, the very opposition between society and technology, as well as their dis-adjustment in terms of speed, do not hold everywhere in the foundational narrative of Software Engineering. In order to understand this point better, it is worth examining how the participant E. E. David describes the process of software growth according to the Garmisch report:

> In computing, the research, development, and production phases are often telescoped into one process. In the competitive rush to make available the latest techniques, such as on-line consoles served by time-shared computers, we strive to take great forward leaps across gulfs of unknown width and depth. In the cold light of day, we know that a step-by-step approach separating research and development from production is less risky and more likely to be successful. Experience indicates that for software tasks ... estimates are accurate to within 10-30 per cent in many cases. This situation is familiar in all fields lacking a firm theoretical base. Thus, there are good reasons why software tasks that include novel concepts involve not only uncalculated but uncalculable risks.
>
> (Naur and Randell 1969: 15 f.)

This complex quotation deserves careful analysis. David focuses here on the pace of software growth. The competition between computer manufacturers forces software professionals to confuse ('telescope') research and production, which should remain separate. Therefore, the uncertainties typical of research (here intended as the

development of innovative software) spread to production. David's metaphor opposes 'leaps' to 'steps'. The leap is for him a dangerous way to move forward, motivated by the lack of knowledge. The step-by-step approach would be a safer way – not, it seems, to slow down the growth of software, but to make the speed of such growth more manageable. One must be reminded once again here that the participants in the Garmisch conference had to face some major doubts concerning large-scale software systems: were such systems actually feasible? In David's terms, the question could have been reformulated as follows: was the speed of software growth actually manageable? Importantly, David attributes the necessity of taking big leaps forward to the lack of a 'firm theoretical basis': in other words, the inability to estimate the feasibility of a software project in a reliable way leads to the impossibility of carrying it out step by step, and ultimately to its failure. The failure of a software project then seems to be related to the failure of the management of time.

According to David, software professionals are fundamentally concerned not just with risk - that is, with the possibility of failure - but also with 'uncalculated' and 'uncalculable' risks. It seems quite understandable that certain risks cannot be calculated due to the lack of accurate knowledge. What is really surprising is David's use of the expression 'uncalculable'. It is not quite common for software professionals and engineers to acknowledge that a technical project involves uncalculable risks. It is worth noting at this point that, although the participants in the Garmisch conference must not have been aware of this, the concept of calculability of time has a distinct Heideggerian echo. As I showed in Chapter One, Heidegger's understanding of technology is deeply connected to his philosophy of time. According to Heidegger (1977), modern technology is a form of calculation, and calculation has its roots in our relation to the future, and in our attempt to determine future possibilities, which we fear precisely because they appear indeterminate. Heidegger describes this process as 'anticipation' or 'concern': our attempt to control (or to anticipate) the uncertainty of the future creates the basis for calculation, or for circumscribing the realm of possible futures. Understood in a broader historical context, this is what Heidegger identifies as the turning of

Western thought into calculation in the modern age. This is also why for Heidegger technology has a central role in defining modernity.[17]

To summarize David's argument, the concept of risk and calculability are both related to the future: estimates are the expression of a calculability of the future, they actually *presuppose* the calculability of the future. As I have pointed out above, it is precisely this faith in the calculability of time, and therefore in the feasibility of software projects, that is put into question in the Garmisch report and in its narrative of the 'software crisis' as the source of technological 'horror'. At this point I want to posit a question: to what extent can the uncalculability lamented by David be linked to the 'unforeseen consequences' that for Derrida – as I emphasized in Chapter Two - are always implicit in contemporary technology? In order to understand this point better, let me briefly recall Derrida's argument here.

According to Derrida, the acceleration of technological innovation in the contemporary world, coupled with the development of information and telecommunication technologies ('tele-technologies'), constitute a 'practical deconstruction' of the instrumental conception of technology (Derrida and Stiegler 2002: 45). It is true that in the contemporary world technological innovation is massively appropriated by multinational corporations and nation states, by means of their 'research and development' and 'defence' departments, and that in that context technological products become obsolescent very quickly and technological innovations are constantly programmed to support economy. But it is also true that, although programmed and neutralized as controlled 'development', technological innovation still gives rise to unforeseen effects. Derrida even propounds that the greater the attempt to control innovation, the more unforeseeable the future becomes. Such unforeseen effects ultimately deconstruct the understanding of technology as a tool as well as the perception of the human as separate from his tools and a master of them. But in what way do the participants in the Garmisch

[17] Although in modernity technology becomes a project of calculation meant to master nature and humanity (what Heidegger calls the 'enframing' of nature and humanity through calculation), modern technology also opens up for us the possibility of radically reconceiving technology itself by making us conscious of the instrumental approach which has characterized our understanding of technology since Aristotle (Heidegger 1977).

conference acknowledge technology's potential for producing unexpected ('uncalculable') consequences? Certainly, they have to deal with the necessity of managing time - namely, the time of software development, and they attempt to produce a 'theory' that comes to terms with the incalculability of the technological future.

To investigate the problem of uncalculability further, it is worth noting how the Garmisch conference report is dominated by a widespread recognition that the ninety-nine per cent of software systems work – as Jeffrey R. Buxton states - 'tolerably satisfactorily' (15). Only certain areas are viewed with concern. Kenneth W. Kolence comments:

> The basic problem is that certain classes of systems are placing demands on us [software professionals] which are beyond our capabilities and our theories and methods of design and production at this time. There are many areas where there is no such thing as a crisis – sort routines, payroll applications, for example. It is large systems that are encountering great difficulties. We should not expect the production of such systems to be easy.
>
> (Naur and Randell 1969: 16)

We already know that the risky 'classes' of systems are large-scale and real-time ones. Nevertheless, this passage seems to take the argument a step further and to relate the uncalculability of software development to certain demands posed by society that go beyond the technological capabilities of the time.[18] In other words, not only do the conference participants feel the pressure of social demands on them; they also feel that software development reaches its point of crisis when society pushes the boundaries of state-of-the-art technology. The question I want to ask at this point is: do these demands come from society or from technology itself? Here I want to make the suggestion that such a question is at work in the whole of the

[18] One might be reminded again here of Stiegler's image of contemporary technology as a device that 'goes faster than its own time'. Stiegler's favoured analogy is that of 'a supersonic device, quicker than its own sound', whose breaking of the sound barrier provokes 'a violent sonic boom, a sound shock' (Stiegler, 1998a: 15).

Garmisch report and that it silently destabilizes the separation between the technical and the social. Actually, it is precisely when dealing with the issue of the responsibility for the technological risk that the conference participants seem to be confronted with the impossibility of separating technology from society. For this reason, I argue that, rather than as a dis-adjustment between the technical and the socio-cultural systems, the 'software crisis' and the beginning of Software Engineering can be understood better within the framework of the mutual co-constitution of the technical and the social. Let me now explain how this co-constitution works in the Garmisch conference report.

Naur and Randell reinforce the argument that society pushes the boundaries of existing technology by lamenting that, at the time of the Garmisch conference, 'one of the main problems was the pressure to produce even bigger and more sophisticated systems' (17). Buxton adds: 'There are extremely strong economic pressures on manufacturers, both from users and from other manufacturers' (18). In fact, Buxton attempts to find reasons for such pressure: for instance, the demand for automated systems of air traffic control is motivated by the rate of the increase of air traffic in Europe. Yet, even this justification remains wholly within the realm of technology, at the expense of the social: more software systems seem to be needed just because more air traffic is needed. We are brought back here to Randell's observation that at the Garmisch conference the triumphant tones of the technical literature of the 1960s were for the first time confronted by software professionals willing to admit that there existed problems in software development, and to question the feasibility of large-scale projects. David Gries, who was assistant professor at Stanford University in 1968 and one of the youngest participants in the conference, in a brief article published in 1989 recalls how the term 'software crisis' was openly discussed in Garmisch for the first time (Gries 1989: 98). In another article published in the same year, also regarding the Garmisch conference, Bernard A. Galler recounts how at the time everyone knew that there was a crisis in the software field, originated by the 'craft mentality' of software professionals. Thus, there was a widespread inability to cope with the development of large and complex software systems and a strong need to begin treating 'the discipline of software as just that, a scientific discipline' (Galler 1989: 97). A few conference participants, such as Andy Kinslow, came from stressful experiences in large-scale

programming.[19] The best-known example of a large-scale project that incurred extra-costs and resulted in serious delays in the 1960s was OS/360, which I have repeatedly mentioned above and which a few years later inspired Frederick Brooks' work, *The Mythical Man-Month* (Brooks 1995).

Unsurprisingly, poignant doubts regarding the feasibility of large software systems are expressed in the Garmisch conference report. For instance, Ascher Opler states:

> I am concerned about the current growth of systems, and what I expect is probably an exponential growth of errors. Should we have systems of this size and complexity? Is the manufacturer's fault for producing them or the users' for demanding them? One shouldn't ask for large systems and then complain about their largeness.
>
> (Naur and Randell 1969: 17)

Opler's passage is intriguingly ambiguous. To understand it better, one must keep in mind that a separation between society at large and technology has been established and maintained throughout this section of the report, and that the relationship between the two has been expressed in terms of the 'impact' of technology on society as well as of the 'demands' of society. At this point, Opler goes a step further and asks whether the responsibility for the rate of the growth of technology must be attributed to the users or to the producers of technology. What I want to argue here is that the undecidability of this dilemma leaves its mark on the field of Software Engineering and especially on its relationship with time. To elaborate on this point, let me now analyse the following passages, which conclude the section 'Software Engineering and Society' by putting forward two possible answers to this dilemma.

On the one hand, the participants in the Garmisch conference seem to acknowledge that risks are implicit in software, and that software fallibility is unavoidable. This is what David and Fraser state: '[p]articularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software

[19] Randell (1979) speaks of Kinslow as 'a recent refugee from the TSS/360 project', whom he remembers 'still almost visibly suffering from the experience' (Randell 1979: 7).

system can be a matter of life and death' (Naur and Randell 1969: 16). On the other hand, some participants claim that risks could be avoided if an appropriate and effective 'theory' of the development of software was produced. From this second point of view, the approach to software development must be 'systematic' (Shaw 1989), and therefore it must become a form of engineering. However, these two points of view are entangled and one does not exist without the other. Thus, the sentence chosen as a conclusion by Naur and Randell is a quotation from Stanley Gill:

> It is of the utmost importance that all those responsible for large projects involving computers should take care to avoid making demands on software that go far beyond the present state of technology unless the very considerable risks involved can be tolerated.
>
> (Naur and Randell 1969: 18)

This quotation might sound like an attempt to discharge the responsibility for technological risk on society. In fact, it requires deeper analysis, since in what way could policy makers evaluate risks that they do not know? Software professionals are the ones who are expected to have such knowledge. A Habermasian answer might suggest that policy makers should be better informed of technological risks and able to discuss them freely. But what Gill is actually saying here is that society shall not make demands that can be met only by exceeding the current state of technology. I argue that this is the 'point of opacity' – as Derrida would have it – of the foundational narrative of Software Engineering.[20] Indeed, it seems to me that the irreconcilability of these two aspects – and therefore the necessity of calculating incalculable risks, and of attributing responsibility for them – is a point where Software Engineering 'undoes itself' precisely at the moment of its constitution. What Gill means here is that society needs to take responsibility for an incalculable

[20] In Derrida's words (1980) a point of 'opacity' is a concept that escapes the foundations of the conceptual system in which it is nevertheless located and for which it remains unthinkable. For Derrida in every conceptual system we can detect a concept that is unthinkable within the conceptual structure of the system itself – therefore, it has to be excluded by the system, or, rather, it must remain unthought to allow the system to exist. A deconstructive reading looks for points of opacity – that is, for points where the system 'undoes itself'. A deconstructive reading of the foundational narrative of Software Engineering therefore needs to ask: what is it that has to remain unthought in order for Software Engineering to exist?

risk. The real problem here is the incalculability of the speed of technological growth - that is, of the rate at which the state of technology is exceeded.

To summarize the first part of my argument, Software Engineering as a discipline with a theoretical foundation is called for in order to avoid the (unavoidable) fallibility of technology – a fallibility that constitutes the risk posed by technology, or, better, technology *as* a risk. This point of opacity suggests that Software Engineering establishes itself *as a theory* of technology by expelling fallibility from technology – but such a fallibility (the unexpected consequences of technology) is intrinsic to technology itself, and is exactly what allows Software Engineering to exist (that is, the reason why Software Engineering is called for). In other words, Software Engineering performs an impossible expulsion of constitutive failure from technology, with this move establishing itself as a discipline. Since such an expulsion is performed through the calculation of time, it can also be said that in Software Engineering the calculability of time is undone in its very constitution. Going beyond Stiegler's concept of the dis-adjustment between technology and society, I also want to suggest that society is instituted in the Garmisch report as that which places risky demands on technology – while at the same time the report declares technology as constitutively fallible, as something that intrinsically incorporates unforeseen consequences. Therefore, the projection of the fallibility on society - that is, on the demands that society poses to technology - is the way in which the conference participants both assume and discharge responsibility for the technological risk: they cannot actually maintain the boundary between technology and society, because this boundary keeps becoming undone. This is why I said earlier that (in Heideggerian terms) Randell's 'horror' is the result of anticipation *plus* the fallibility of technology. In a way, it can be said that, contrary to Heidegger's understanding of the relationship with death as constitutive of a temporality which is more 'authentic' than the temporality of calculation, in Software Engineering the question of death (for instance, the death of New York's inhabitants caused by a ballistic device gone wrong) is dealt with *as* a problem of calculation.

In order to understand how the calculation of time is performed in Software Engineering, let me now examine what is meant by the 'systematic approach' to

software that, as we have seen, the participants in the Garmisch conference recommend. First of all let me go back for a moment to the quotation regarding the choice of the term 'software engineering' by the NATO Science Committee as the title of the Garmisch conference:

> The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.
>
> (Naur and Randell 1969: 13)

This passage traces an interesting parallel between engineering and 'theory'. The expression 'more established branches' refers to construction engineering - that is, bridges, buildings and any other 'hard' kind of construction (Naur and Randell 1969: 17). As I have emphasized earlier on in this chapter, Software Engineering is both a 'provocative name' and the expression of the need for a theoretical foundation of software development. The use of 'engineering' as a synonym for 'theoretical foundation' is quite striking, especially since engineering is here regarded as dealing with buildings and with what can be generally considered 'hardware'. Actually, throughout the report the editors and the conference participants point out that software engineering 'is in a very rudimentary stage of development as compared with the established branches of engineering' (Naur and Randell 1969: 17). For instance, Doug McIlroy observes:

> We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontation with hardware people because they are the industrialist and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries.
>
> (Naur and Randell 1969: 17)

This passage contextualizes the need for software engineering within the wider picture of industrialization. To understand this point better, it is worth having another look at the software scene of the 1960s. At the time, thinking of software as

a possible object of industrialization was not a complete novelty. In fact, by 1968 the fact that 'software was an important commodity in its own right' was widely recognized (Randell 1979:2). Just in the United States there were around five hundred organizations concerned with selling and/or producing software – albeit the term 'software' had made its appearance in normal business parlance only a few years earlier (2). Nevertheless, the process of developing software was not well understood yet. In an article published in 1989, Galler explains that in the late 1960s software professionals needed to begin to study software development in the hope that it could be 'formalized and controlled' and that this would in turn raise the attention of both the computer industry and the university (Galler 1989: 97). For this reason McIlroy presented a paper on software production, entitled '"Mass-Produced" Software Components', at the Garmisch conference. Published among the appendices to the conference report, this paper investigates 'the prospects for mass-production techniques in software' and recommends the creation of software components according to the same criteria that regulate the production of hardware components. For McIlroy, one important reason for the weakness of the software industry in the late 1960s was the absence of a software components sub-industry (Naur and Randell 1969: 139). With formidable insight he wrote:

> I would like to see components become a dignified branch of software engineering. I would like to see standard catalogues of routines, classified by precision, robustness, time-space performance, size limits, and binding time of parameters. ... I want to have confidence in the quality of the routines. ... What I have just asked for is simply industrialism, with programming terms substituted for the more mechanically oriented terms appropriate to mass production. I think that there are considerable areas of software ready, if not overdue, for this approach.
>
> (Naur and Randell 1969: 150)

It is not important here to understand in detail what a routine is and what the different associated parameters mean. For the scope of my argument, a routine can be regarded as a small independent portion of code that can be used as a building block by software professionals to construct their own systems, in a standardized

way and with acceptable confidence in its reliability. It must be kept in mind that what are now commonly known as software libraries - that is, collections of routines that perform an immense number of simple tasks - did not exist in the 1960s.[21] What is really significant in the above passage is that software is viewed as a product and software production is approached as a formalizable – and to some degree even automatizable - process. McIlroy's paper makes obvious how in the late 1960s the concept of engineering was introduced into the software field in connection with the view of software as a commodity and an industrial product. However, such standardized production could not be accomplished unless the process of software development was well understood, and the Garmisch conference focused precisely on pursuing such an understanding.

Interestingly, in the Garmisch conference report the aircraft industry serves as a term of comparison for the status of software industrialization. In particular, Ronald Graham of Bell Labs maintains that '[t]oday we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes – build the whole thing, push it off the cliff, let it crash, and start over again' (Naur and Randell 1969: 17).[22] Graham's passage, which institutes a parallel with the aircraft industry and its supposed initial inability to anticipate the future, displays a certain confusion between the conceptualization of a software system and the understanding of its development. As I will show in a moment, such confusion returns over and over again throughout the Garmisch conference report. In fact, while a software system is mainly envisaged as a set of interrelated components that work together in order to achieve some objective, its development is understood as the process through which such a system is constructed - or better,

[21] Examples of routine libraries commonly used today are Java and C++ libraries. In fact, their availability has changed the whole technique of programming. In the past, programmers used to write code that accomplished a certain task on a certain machine, starting from scratch and following a logical structure known as 'algorithm'. Today they are more likely to figure out what routines they need to perform a given task on a given machine and to look for them on the Internet.

[22] However, Mahoney remarks that '[h]istorians of technology know that the Wright Brothers' successful flight was in fact the culmination of a carefully planned, theoretically and empirically informed, program of research and development. In particular, they had a relatively clear idea of what problems they had to solve and of how they might go about solving them. Whether or not their approach might have served as a useful example for fledgling software engineers, it does not seem *prima facie* to constitute a negative example' (Mahoney 2004: 9).

as the conference participants commonly say, 'written'. Although the report focuses on understanding and defining the process of software development, the system and its development are never clearly separated. For instance, in Graham's passage the failures of the process of development (in which investment and time spin out of control) are caused by the poor understanding of the system in the first place. I want to argue here that the distinction between the software system and its development – that is, between process and product – is another 'point of opacity' of Software Engineering since, albeit necessary, it cannot be kept up at all times. Even more interestingly, albeit so far the report has depicted software growth as a problem *per se*, in a subsequent passage Fraser states:

> One of the problems that is central to the software production process is to identify the nature of progress and to find some way of measuring it. Only one thing seems to be clear just now. It is that program construction is not always a simple progression in which each act of assembly represents a distinct forward step and that the final product can be described simply as the sum of many sub-assemblies.
>
> (Naur and Randell 1969: 17)

Here Fraser seems to focus on the inordinate character of software growth, and particularly on the fact that the process of development is not straightforward - in his words, it is not a 'simple progression'. For him, software professionals need to understand the complexity of the process of software development, but at the same time they need to make it more linear. Or even better, they must acknowledge that such a process is not in fact linear, but at the same time they have to avoid backward steps at all cost. An ambivalence can be detected here in the conceptualization of software growth, which is considered a risk (since it implies failures) but at the same time a proof of success (and therefore it must be maintained by avoiding backward steps). Quite clearly, even more than the pressure of society, it is the implicit fallibility of software that worries conference participants. Also, what Fraser puts forward here is the need for a linearization of time – that is, the need to measure 'progress' within the temporality of software development, which is not always 'a simple progression'. The very concept of progress implies the movement toward an objective, thus calling for a linearization of time. In Chapter One I

pointed out how the linearization of time, according to Derrida and Leroi-Gourhan, plays an essential part in the alphabetization of non-linear writing. The model of the line is the basis which allows for the reduction of writing to a mere instrument for the recording of speech – thus, it also becomes the foundation of instrumentality. Here Fraser's appeal to the linearization of time is part of the general attempt to think software in instrumental terms.

Accordingly, in this first section of the report the focus shifts gradually from 'engineering' as a disciplinary model toward its specific approach to time: engineering involves planning - that is, the calculation of time - and systematic thinking. This view is supported by comments offered two decades later by numerous scholars. For instance, in her 1989 article entitled 'Remembrances of a Graduate Student' Mary Shaw recalls 1968 as a 'memorable year' and explains: 'it was the year the software research community really started thinking systematically, even formally, about software structures' (Shaw 1989: 99). She also recounts how the most relevant question at the time was: 'why shouldn't software be a product, subject to analysis, prediction, and organized development?' (99), and claims that the Garmisch conference represented a turning point in the passage from *ad hoc* to systematic practice (100). In fact, in 1968 the very concept of software acquired a totally new meaning, since the Garmisch conference report 'played a major role in shifting our thinking from *programs* to *software*' (99). Thus, the term 'software' itself became the carrier of the idea of systematicity and of the calculation of time.

To summarize the above argument, Software Engineering makes its appearance in the context of the early industrialization of software production and in opposition to the concept of craftsmanship, and it relies on a model of engineering as the calculation of time. The main object of Software Engineering as a discipline is the calculation of the time of software development in order to industrialize it. This calculation is to be realized through the formalization of the process of software development, and the Garmisch conference report contains a number of diagrams that were proposed at the conference in order to attempt such formalization. To understand this point better, let me refer here to the diagram offered by Calvin Selig (Naur and Randell 1969: 21), which I reproduce in Appendix B. This image is particularly interesting because it makes explicit reference to the concept of

'documentation' - whose significance I will highlight in a moment. Moreover, it can be easily understood even without analysing all the technical terms it deploys.

Selig's graphical representation of the process of software development makes it clear right from the start that such a process does not necessarily imply linear progress. Naur and Randell also remark that, as soon as this draft was presented at the Garmisch conference, Perlis amended it by suggesting that a feedback loop be added to it 'for monitoring of the system' (Naur and Randell 1969: 19).[23] It is worth noting here that, with little variation, Selig's scheme has become the typical representation of what is now called the 'software life-cycle'. However, according to Selig, the process of software production is the sequence of different stages, where each stage can be described as a period of intellectual activity which produces a usable product that constitutes the point of departure for the subsequent stage. At the beginning, a 'problem' is identified to which the software system will be the answer. At this point in the report, it must be noted, software has shifted from being a 'problem' (Naur and Randell 1969: 3) to being a solution, while the problem has been relocated to the external world. This again has become part of the methodological core of Software Engineering: there are problems *out there* that software helps to solve. The first stage of intellectual activity, named 'analysis', produces a description of the problem – which is substantially a document written in a natural language. The stage of analysis is variously enmeshed with the phase of design (the two terms are used quite interchangeably in the report), which constitutes a refinement of the problem description in order to propose a system that solves it. The phase of design produces a complete system specification - that is, a stable description of *what* the software system is supposed to do. The system specification is also the point of departure for the stage of implementation, which basically determines *how* the system will do what it is supposed to do. It is immediately apparent from Selig's picture that the boundaries between these stages are quite blurred. What is not obvious – although it emerges rather clearly from the

[23] As I showed in Chapter Two, a feedback loop is a process that circulates the output of a system back into the system as input. Although the idea of a system that governs itself by means of such a process dates back to the Greeks, the concept of the feedback loop found its broader theoretical development and practical applicability in first-order cybernetics during the 1930s and 1940s. Here I want to point out that, in the context of software development, the feedback loop also represents a temporal relationship between two (supposedly separated) stages of the process. I will return to this point later on in this chapter.

totality of the Garmisch conference report - is that the terminology referring to the different stages of software development is also remarkably unstable.[24] The only clear separation is between the 'what' and the 'how': one must define 'what' a system is supposed to do *before* developing it, and the development of the system is the determination of 'how' the system does what it is supposed to do. Yet, even this distinction is put into question later on in the report, and the boundary between the 'what' and the 'how' – as I will show in a moment – turns out to be quite difficult to maintain. However, the stage of implementation 'translates' (Selig's term) the specification of the system into the 'actual' system (the 'working system') - that is, a software system that can be installed on a computer. Before it is delivered to the final user, the system needs to be tested – namely, it must be verified whether the system actually does 'what' it is meant to do and whether it does it 'how' it is meant to do it. This stage is variously called 'testing', 'acceptance' or – in Selig's case – it can be incorporated into a general stage of 'maintenance', which comprises all the activities performed on a completely operational software system, such as corrections and modifications that might be carried out even after the system has been delivered to its users. As soon as it becomes operational, the system starts to become obsolete – and it will finally be abandoned in favour of newer systems.

Two points must be emphasized here: firstly, as I have pointed out above, Selig's image explicitly refers to 'documentation' – that is, the body of written texts produced in the course of the whole process of system development. Secondly, software development is defined as a process involving much more than the mere writing of computer programs; therefore, it extends over a longer time span. As for the first point, it is worth noting that the definition of 'documentation' will remain an open problem of Software Engineering for decades, and that the term will continue to shift from 'user documentation' - that is, all the manuals and guidelines provided to final users as an explanation of the software system - to 'technical' or 'internal documentation' - which functions both as a description of the system and as a means of communication between software developers. Regarding the second point, Selig comments that, although programming has traditionally been viewed as

[24] From Selig's picture it is also clear that, since every single term used so far has acquired a specific meaning as a stage of the total process, the only term left that is generic enough to indicate the whole process is 'software-system building'.

the production of code, in practice programmers perform their activities over the time span that goes from the moment when the 'problem' has been understood to the moment when the system becomes obsolete. Both of these points concur in elucidating how the formalization of the process of software development is achieved in Software Engineering. In order to control it, the process of software development is broken up into periods of activity, each of which produces, as a result, a piece of writing. Each piece of writing is the point of departure for the following phase, and it is supposed to be used as a tool for developing more pieces of writing. Thus, the organization of time is carried out through a practice that I call 'writing' because this is the term used throughout the Garmisch conference report and because it produces written texts – although the nature of such 'writing' needs to be explored further. Some of these written texts are called 'documents', some are called 'software' or 'programs' or 'code', and these terms keep shifting. Moreover, the written text resulting from the stage of analysis - namely, the description of the problem - constitutes the point of departure for the whole process of software development. At the same time, by describing the problem as something pre-existing 'out there', this document also constitutes a narrative of the origin of the software system itself. More precisely, it projects a 'problem' in the world to allow the software system to become its 'solution' – or, it can even be said, it performs the expulsion of the software system *as* a problem in order to justify its very existence as a solution. By doing this, Software Engineering also constitutes software as 'instrumental' – that is, as the means by which the goal of solving a pre-existent problem can be reached. At the same time, though, software escapes instrumentality because the distinction between problem and solution does not hold. As I have just pointed out, software is both problem and solution – indeed, it emerges at the point where the distinction between the two becomes undone and it exists only as the precarious stabilization of this distinction. Let me now expand on this point for a little while.

Further on in the Garmisch conference report, Willem L. Van der Poel formulates the following question:

> The specifications of a problem, when they are formulated precisely enough, are in fact equivalent to the solution of the problem. So when a

> problem is specified in all detail the formulation can be mapped into the solution, but most problems are incompletely specified. Where do you get the additional information to arrive at the solution which includes more than there was in the first specification of the problem?
>
> (Naur and Randell 1969: 52)

This quotation shows the difficulty of describing the transition from problem to solution purely in logical terms - that is, in terms of the logical 'completeness' (or exhaustiveness) of the description of the problem. Interestingly, Dijkstra answered Van der Poel's question by comparing his own experience as a computer sciences teacher to that of a teacher of composition at a music school. One cannot teach creativity, he claimed, nor can one ensure that one gets thirty gifted composers out of thirty pupils. What a teacher can do is to make pupils 'sensitive to the pleasant aspects of harmony' - but 'the rest they have to do themselves' (52). Thus, Dijkstra resorts to individual creativity as an explanation for the passage from problem to solution in the process of software development. As I explained in Chapter Two (and as Derrida suggested in his essay of 1989, 'Psyche: Inventions of the Other'), what is beyond ('in excess of') a procedural method is traditionally ascribed to 'genius' or 'inspiration', and therefore recuperated on the level of subjectivity. Intriguingly, Dijkstra does so precisely in the context of a paper where he attempts to establish some 'objective' measurement for the logical completeness of the description of a software system. Dijkstra's move is particularly relevant here because it shows how the transition between problem and solution gives rise to a conceptual impasse. I want to suggest that this impossible transition between problem and solution actually masks the expulsion of the problem from the process of software development in order to establish a narrative of the origins of the software system. Even more importantly, by separating problem from solution, while subsequently relating them through a series of written texts, the Garmisch conference report invests 'writing' with a central role in the organization of the time of software development.

To recapitulate the above argument, Software Engineering (as a method for industrializing the production of software) involves the organization (and linearization) of time through 'systematic thinking', and such systematicity involves

the control of the process of software development through practices of writing, as well as through the establishment of a pre-existing 'problem' that justifies the existence of the process of software development itself. However, it remains to be established in what way the different written texts produced in different stages of software development take different - albeit shifting - forms and names, and how they relate to each other and to what the Garmisch conference report names as 'software'. Let me expand on this point.

In order to examine how different pieces of writing are produced in the course of software development, I want to focus now on the first of these texts – namely, the so-called external specifications (or 'specifications' *tout-court*) of the software system – and on its relationship with the text produced subsequently – that is, the 'internal design' (or simply 'design') of the system. Selig writes:

> External specifications at any level describe the software product in terms of the items controlled by and available to the user. The internal design describes the software product in terms of the program structures which realize the external specifications. It has to be understood that feedback between the design of the external and internal specifications is an essential part of a realistic and effective implementation process. Furthermore, this interaction must begin at the earliest stage of establishing the objectives, and continue until completion of the product.
>
> (Naur and Randell 1969: 22)

There are many points of interest in this quotation. What has been characterized earlier on in the report as the 'how' and the 'what' of the system – or, in Selig's terms, 'analysis' and 'design' - is renamed here as the 'external specifications' and 'internal design', at the same time establishing a feedback loop between the inside and the outside of the software system. Selig's passage implies a spatial representation of the software system, whose inner and outer parts are meant to be 'described' by written texts. Furthermore, the external part of the software system is defined in terms of 'availability' (to the user) and 'control' (by the user). Importantly, both availability and control belong to the terminology of instrumentality, thus describing the software system as a tool. The internal design is

defined as a text that provides a description of the programs - that is, of the single pieces of software - that will compose the software system as a whole. Selig uses here the term 'to realize': for him the specifications describe the system, while design and code 'realize' it.

Numerous remarks are made during the Garmisch conference report on the way in which such 'realization' must take place. In his paper entitled 'On the interaction between software design techniques and software management problems' Kolence writes:

> A software design methodology is composed of the knowledge and understanding of what a program is, and the set of methods, procedures, and techniques by which it is developed. With this understanding it becomes obvious that the techniques and problems of software management are interrelated with the existing methodologies of software design.
>
> (Naur and Randell 1969: 24)

According to Kolence, the realization of the software system involves the 'interaction' (editors' term, 24) between software 'design techniques' and software 'management problems'. This complex quotation hints again at the difficulty of distinguishing between the software system and the process of its development. Moreover, in any software project, the technical aspects of software development – which Kolence names 'design' - cannot be clearly separated from the organizational aspects – or 'management'. And yet, some kind of separation is necessary if Software Engineering is to have a methodology – which is, one must recall, precisely what constitutes the difference between craftsmanship and engineering. Considering this transition of software development from craftsmanship to a process governed by a methodology, Alexander d'Agapeyeff insists that 'programming is still too much of an artistic endeavour' (24). His conference paper, significantly titled 'Reducing the cost of software' invokes 'a more substantial basis' for the monitoring of the process of software production. A monitoring methodology should substantially 'make software visible' in terms of programs, of 'the flow of their execution', of the 'shaping of modules', of their testing

environment, and finally of the 'simulation of run time conditions' (24). Once again, it is not necessary to understand the technicalities of this passage. What must be pointed out is that d'Agapeyeff clearly recognizes that the only part of the software system and of the process of its development - or, even better, of the software system *in* development - which is actually visible is the part that is put in writing. Thus, during its development, the system is made visible through the written texts that mark the successive stages of such development.

However, it is worth noting that d'Agapeyeff's passage uses the term 'artistic' as a pejorative term. I would translate 'artistic' here as 'unmonitorable', 'unmanageable', or, even more precisely, 'not reducible to a methodology'. The term even hints at an excessive individuality of the activity of programming. Regarding d'Agapeyeff's paper, Kinslow comments:

> There are two classes of system designers. The first, if given five problems will solve them one at a time. The second will come back and announce that these aren't the real problems, and will eventually propose a solution to the single problem which underlies the original five. This is the 'system type' who is great during the initial stages of a design project. However, you had better get rid of him after the first six months if you want to get a working system.
>
> (Naur and Randell 1969: 24)

Kinslow claims here that different mentalities are required by different stages of software development – and that an idiosyncratic approach to software, however brilliant, can put the whole process at risk. This brings us back to the moment when Dijkstra's takes recourse to individual creativity in order to explain how software development progresses. Dijkstra's and Kinslow's positions interlock in a rather problematic way. On the one hand, individual creativity is evoked as the mysterious agent that allows for the translation of problem into solution, and such translation is depicted as a form of the unexpected – analogous to the unforeseeable and inexplicable leap that turns a student of music into a talented composer. On the other hand, the view of technology proposed by Kinslow in the above passage, and quite likely shared by his colleagues at the Garmisch conference, attributes the

emergence of the unexpected to the human mind: too much genius (or too many unexpected consequences) can be dangerous, therefore it must be kept under control, and a good project manager assigns the appropriate practitioners to the appropriate stage of the software project in order to keep it manageable. Thus, in the process of software development the emergence of the unexpected seems to be both the propeller of development itself *and* what puts development at risk. Moreover, the transition from problem to solution is represented as a 'good' form of the unexpected – that is, as an unforeseeable creative leap that, although it cannot be anticipated, is nonetheless manageable. The 'excessive' creativity of the 'system type' is in turn portrayed as 'bad' unexpected – something that exceeds the management of the project and threatens it. But, as Derrida would have it, the unforeseeable that can be anticipated (or managed) is not unforeseeable at all. Therefore, once again we are faced with the 'point of opacity' of the unexpected consequences of technology and with the impossible expulsion that Software Engineering tries to perform: while the aim of Software Engineering claims to be the expulsion of the unexpected from technology, the unexpected – represented as a creative leap – is also acknowledged as constitutive of technology. Thus, the question about the different kinds of texts produced during the process of software development can be reformulated as such: what is the relationship between these different texts and the unexpected? Could it be that such texts are different precisely because they entertain different relations with the unforeseeable effects of technology?

In order to explain this point, let me now turn to the analysis offered by J. A. Harr in his paper, 'The Design and Production of Real-Time Software for Electronic Switching Systems', a part of which is reproduced in the report (Naur and Randell 1969: 25). Harr's paper describes the connections between the process of software development and the organization of the correspondent project group, thus attempting to clarify the way in which the technical characteristics of the software systems and the organizational aspects of the working group mirror one another. Harr breaks down the design and production process into thirteen steps, starting from the extensive specification of all the system functions up to the final global testing of the system after it has been completed. He then proposes a structure of the programming group that reflects the structure of the system by allocating specific

groups of programmers to each of its parts. Here Harr makes a significantly innovative point about the structuring of both software systems and the groups or organizations that produce them. To understand its relevance, let me turn once again to the context of the 1968 software scene. The literature of the time showed a growing interest in developing techniques and practices of system structuring. The so-called 'modular programming' was already in vogue (Randell 1979: 4) – that is, an approach that allowed software developers to break down a complex system into different sub-systems, each of which was supposed to perform a certain function and whose combination achieved the general purpose of the system. Each sub-system could in turn be broken down into smaller components, until reaching a level where the whole system was decomposed into manageable parts - each a relatively independent piece of software that could be developed by single programmers separately. As Randell comments, in the late 1960s software professionals were increasingly developing 'a belief in the need to "divide and conquer" system complexity' (4). For instance, at the time H. T. Hicks wrote: '[i]n general, large problems are most easily solved by factoring them repeatedly into smaller, more logically independent parts until the solution of each part is either available (i.e. the problem has been solved before and the result recorded) or is clear. The set of solutions thus developed forms the solution of the large problem' (Hicks 1968: 52). On the same point, in a paper published in the same year as the Garmisch conference, Larry Constantine advanced the hypothesis of a 'structural theory' of programming based on writing: '[a] program is an ordered set of statements and aggregates of statements defining, describing, or directing the performance of some task. ... The aggregates of statements are called modules, and a module is a program' (Constantine 1968: 15). This passage is extremely important, because it presents a definition of what a 'program' is, and it relates it to the concept of module. According to Constantine, a module is a program, which in turn is an aggregate of statements written in a programming language. Consistently with Hicks' passage, modules are defined as manageable parts of a broader software system, which perform a task. It is important to highlight how ideas such as Constantine's and Hick's were rarely mentioned in the academic computing literature of 1968. Quite shockingly if compared with today's state of the art, even

the term 'structured programming' had yet to be invented and turned into a catch phrase (Randell 1979: 5).[25] Most importantly, Conway had already published his brilliant paper titled 'How Do Committees Invent?', where he formulated what later became known as Conway's Law - that is, that 'organizations which design systems are constrained to produce designs which are copies of the communication structures of the organizations' (Randell 1979: 5). According to Conway,

> [a] contract research organisation had eight people who were to produce a COBOL and an ALGOL compiler. After some initial estimates of difficulty and time, five people were assigned to the COBOL Job and three to the ALGOL Job. The resulting COBOL compiler ran in five phases, the ALGOL compiler ran in three.
>
> Two military services were directed by their Commander-in-Chief to develop a common weapon system to meet their respective needs. After great effort they produced a copy of their organisation chart.
>
> (Conway 1968: 30)

This being the context, it comes as no surprise that in the Garmisch conference report the overall terminology continues to shift between 'documentation', 'software', 'module', 'program', 'code' and 'writing'. Actually, although the difficulty of breaking down the process of software development and of naming its different stages are extensively discussed in the report, the conference participants seem less aware of the continuous switches and transpositions in the names of the texts produced in each of these stages.

In sum, the instability of the terminology used to indicate the different pieces of writing produced during software development must be understood in the context of the state of the art of programming in the late 1960s. However, the differences between texts seem to be related to their belonging to different stages of software development – that is, to the organization of time within a specific project, which in

[25] Dijkstra's well known paper 'Notes on Structured Programming' was written in August 1969. However, at the time of the Garmisch conference, his letter 'Go To Considered Harmful' – which famously inaugurated the age of structured programming – as well as the first responses to it - had already been published (Dijkstra 1968).

turn is mirrored by the structure of the project group. To understand these differences better, let me now turn to the transition between the texts produced in the phase of design (which are generally written in some kind of formalism combined with natural language) and 'code' – which is the outcome of the stage properly called 'production' (or 'implementation').[26]

The editors of the Garmisch conference report emphasize the major challenge of distinguishing between 'design' and 'production' (Naur and Randell 1969: 30) – a distinction contested by several conference participants. Therefore Naur and Randell feel the need to clarify how 'production' is 'not just the making of more copies of the same software package (replication), but the initial production of coded and checked programs' (31). Clearly, although the use of the term 'production' can be related to the increasing industrialization of software production, as a synonym for 'implementation', it does not indicate the mere mechanical process of making copies of a program, but the much more complicated passage from system specification to code.[27] For instance, in his conference paper on the concept of software production Naur writes:

> Software production takes us from the result of the design to the program to be executed in the computer. The distinction between design and production is essentially a practical one, imposed by the need for a division of the labor. In fact, there is no essential difference between design and production, since even the production will include decisions which will influence the performance of the software system, and thus properly belong in the design phase. For the distinction to be useful, the design work is charged with the specific responsibility that it is pursued to a level of detail where the decisions remaining to be made during production are known to be insignificant as to the performance of the system.
>
> (Naur and Randell 1969: 31)

[26] Today implementation has become the commonly used term for the coding phase of a software project. Anyway, this use of the term 'production', and the related ambiguities, are due to the original organization of the Garmisch conference, which was divided into the major thematic areas of 'Design', 'Production' and 'Service'.

[27] It is worth mentioning that in the late 1960s the method for producing copies of software was not as consolidated and secure as it is today, neither was it completely automatized. In fact, it involved a lot of manual activities and it could frequently introduce errors in the piece of software being copied.

In this extremely important passage, Naur denies any essential difference between design and production. For him the decision to break down the process of software development in different stages marked by different pieces of writing has been made simply in order to favour the division of labour.[28] I want to emphasize how Naur in fact introduces a very subtle point here: the level of detail of the system description produced at the stage of design has the predominant goal of maintaining such a division of labour. More precisely, he seems to imply that an appropriate level of detail in design would completely erode the space left for implementers to make decisions of their own, and would thus destroy their ability to influence the software system in any way. The detail given in the piece of writing produced during the stage of design ultimately constitutes the means for the control of the workforce. It is even clearer now why for d'Agapeyeff - as I have shown earlier on in this chapter - a less and less systematic view of the software system – a kind of blindness, almost, or at least the selective forgetting of parts of the system - seems to be preferable at the later stages of the project in order to limit the risks of excessive 'creativity'.

But the most important point that can be evinced from Naur's observation is that there is actually no essential difference between design and production. Thus, it can be argued that there is no difference between the written texts produced in these two stages - that is, between design documents, which, as I have pointed out above, are written in natural language and accompanied by formal notation, and computer programs. In fact, Naur rejects the existence of any intrinsic difference between what Hayles (2005) calls 'writing' and 'code'. The only difference that Naur acknowledges is the introduction of a foreclosure - namely, at the level of design the 'practical' division of labour forecloses the possibility of decision at later stages of software development. What is already inscribed, or written down, at the stage of design cannot be changed at the stage of production; it cannot be decided upon any more; it cannot be undone, unmade; it is subtracted from the process of decision, of change, of further inscription. Writing performs a foreclosure of time through what

[28] The very distinction between hardware and software, which dates from John von Neumann, and his understanding of the general structure of computers, is perhaps the most famous example of a technical decision that led to a division of labour (Bolter 1984).

is already inscribed - that is, the 'level of detail' of the design documentation. By foreclosing the possibility of a decision, writing also forecloses responsibility in the stage of production. Finally, it prevents feedback from production into design, from the 'how' into the 'what' – a feedback that is, nevertheless, necessary and even unavoidable, as I have explained earlier on in this chapter. Thus, the differentiation of the practice of writing in software development is an attempt at foreclosing an unavoidable iteration. For this reason, such a foreclosure is destined to fail – and yet, the process of software development relies on it and tries to preserve it at all times. The 'opacity' of the distinction between 'writing' and 'code' becomes apparent here: if it did not progress from specifications to code, the system would never be realized. And yet, there is no 'essential' difference between the two.

To understand this point better, let me now examine two important remarks made in the Garmisch conference report about the 'dangerous' nature of the distinction between design and implementation (Naur and Randell 1969: 31). The first is Dijkstra's critique of such a 'rigid separation', which for him puts at risk the 'correctness' of software. One must be reminded here that the correctness of a piece of software is defined as its compliance with what the specifications say that it will do. In other words, correctness concerns the 'what' of the system. Dijkstra argues that '[w]hether the correctness of a piece of software can be guaranteed or not depends greatly on the structure of the thing made' (31). Thus, for him, up to a certain point the 'what' relies on the 'how' – or, better, the 'how' needs to be constantly monitored against the 'what'. Not surprisingly, once again the distinction between design and implementation, documentation and code, the 'what' and the 'how' does not hold. In a way, it can be said that the 'what' can exist only as a 'how', that design can be realized only as implementation and documentation only as code.

The second – and related - remark is Kinslow's. For him, besides being hard to break into clearly separate stages, software development *must* involve a certain degree of iteration – that is, software cannot be developed without a process of iteration. It is worth quoting him at length:

> The design process is an iterative one. … In my terms design consists of:
>
> 1. Flowchart until you think you understand the problem.
> 2. Write code until you realize you don't.
> 3. Go back and re-do the flowchart.
> 4. Write some more code and iterate to what you feel is the correct solution.
>
> If you are in a large production project, trying to build a big system, you have a deadline to write the specifications and for someone else to write the code. Unless you have been through this before you unconsciously skip over some specifications, saying to yourself: I will fill that in later. You know you are going to iterate, so you don't do a complete job the first time. Unfortunately, what happens is that 200 people start writing code. Now you start through the second iteration, with a better understanding of the problem, and it is too late. This is why there is version 1, version 2, version N. If you are building a big system and you are writing specifications you don't have the chance to iterate, the iteration is cut short by an arbitrary deadline. This is the fact that must be changed.
>
> (Naur and Randell 1969: 32)

In this long passage Kinslow suggests his own solution for the difficult problem of breaking down software development into steps. It is not important here to fully understand what a flowchart is; what matters is that Kinslow emphasizes once again the 'iterative' nature of the understanding of the software system (here again understood as 'the problem'). What I want to point out is that for Kinslow software professionals understand the system *through repeated attempts to build the system itself*. Drawing a flowchart is a first attempt to understand the system. On the basis of this flowchart, some code is developed which constitutes a second, further attempt to understand the system. At a certain point in time, the person writing the code (which may or may not be the one who drew the flowchart) meets some unexpected obstacle – typically, they realize that the flowchart contains some inconsistency. Therefore, the second understanding of the system feeds back into the first, causing it to be repeated differently and to produce a third understanding of the system (again in the form of a flowchart) – and so on.

Not only is the process of understanding presented as a matter of 'making visible' here (for instance, code makes visible some inconsistencies or errors in the flowchart), but also, if the person going through the iteration cycles is one and the same, then their understanding of the system - what in philosophical terms we might call their 'consciousness' - develops *through* the inscription of marks (flowcharts, code). In other words, the very process of the exteriorization of the system through writing makes the system understandable. To understand this point better, it is important to recall how, in the tradition of originary technicity, technology is understood as 'the support for the inscription of memory' (Hansen 2004: 3). In such a tradition, technology is always related to memory, because any technical instrument registers and transmits the memory of its use. For instance, as Stiegler argues, a carved stone used as a knife preserves the act of cutting, thus becoming a support for memory. In this sense, technology is the condition of the constitution of our relation to the past. It is only through memory that human beings gain access to their own past, and therefore become aware of themselves - that is, gain consciousness (Stiegler 2003b). According to Stiegler, it can also be said that human being 'exteriorize' their memory into technological objects, which in turn are nothing but memory exteriorized. Also, functioning as a support for memory, a technical object 'forms the condition for the givenness of time in any concrete situation' (Hansen 2003: 3). From the point of view of originary technicity human beings can experience themselves only through technology. I would go so far as to say that the different kinds of inscription illustrated in Kinslow's passage are the way in which human consciousness constitutes itself in relation to software. The question I want to ask at this point is: what is the relevance of looking at the different stages of software development, as well as at the texts produced in these stages, as parts of a process of exteriorization – that is, as part of the constitution of time and consciousness? And, first of all, in what way does this understanding of software development relate to the unexpected consequences of technology that I have examined earlier on in this chapter? To answer this question, let me analyze Kinslow's argument a little further.

As I have suggested above, a corollary of Kinslow's argument is that iteration puts the unforeseen consequences of technology at work in order to control technology: for instance, an inconsistency in the flowchart becomes unexpectedly apparent in code, in turn allowing for the modification of the flowchart itself. In fact, this example shows how unpredictability works at a very subtle level in software production - that is, it plays a pivotal role in enabling the passage between its different stages. However, although in this passage Kinslow argues in favour of iterability, he also acknowledges that in large projects which involve a huge number of participants it is quite difficult to iterate the process of design. Drawing on his own experience, Kinslow emphasizes that one tends to 'skip' parts of the specification because one 'unconsciously' postpones the understanding of such parts to the future. However, the subsequent development of code make visible this gap in understanding – that is, it makes visible the unanticipated consequences of the software system which is being developed. Hence the 'danger', or risk, implicit in breaking down the process of software development into separate stages – danger which is nevertheless unavoidable if software development is to progress from specifications to code.

In sum, the different pieces of writing (or better, the different forms of material inscription) produced in the process of software development - from specifications written in natural language to code - differ only in terms of their foreclosure of time, but such a foreclosure is ultimately impossible. The unexpected consequences of software cannot be avoided in the last instance. Indeed, they need to exist in order for software to exist, but they also must be excluded - continually and strenuously - in order for software to reach points of stability (for instance, its existence in the form of specifications, or flowcharts, or code).[29]

[29] It is worth noting that Kinslow's passage shows that the expression 'to write code' was very common in the late 1960s. For instance, Kinslow states that, if different people are involved in the process of iteration, different processes of writing start in different moments, drawing on different (and differently flawed) points of departure, which leads to the development of different versions of the software system. It is quite obvious from the Garmisch conference report that in 1968 the use of the term 'writing' in relation to software was largely accepted, although writing had still to be theorized as a way to make software visible – that is, as I have explained above, to understand it - and thus to control its development.

Let me now explore the relationship that the different texts produced in software development – and in particular code - entertain with time and the unexpected at some length. To do so, it is worth analysing the following quotation from Ross:

> The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from. The projects that are called successful, have met their specifications. But those specifications were based upon the designers' ignorance before they started their job.
>
> (Naur and Randell 1969: 32)

This complex passage deals with the central problem of software development, and therefore of Software Engineering - namely, the impossibility of following a sure path in which a system is completely and exhaustively specified *before* it can be realized. This quotation shows that, from the very beginning, software professionals had a very clear view of this 'paradoxical' aspect of Software Engineering. Apparently here Ross discusses the fact that, as Kinslow also argues, the specification of a system is never complete and that inconsistencies become clear only later on, when the process of implementation starts.– That is, inconsistencies are made visible by implementation, thus requiring an iterative improvement, – or rewriting, of specifications. In fact, what Ross is saying here is that the sequencing of the process of software development over time, which is the basis of Software Engineering, is impossible *as such*. One invariably starts doing what one wants to do before knowing what it is – this is how I want to reformulate Ross' statement here. This is where a paradox lies: a project is successful because it meets its specifications - that is, it does what its specifications say it must do - but the specifications were written when one did *not* actually know what the system was supposed to do. How can something based on the lack of knowledge be realized successfully? In this paradox the mutual constitution of time and technology becomes apparent. Not only does one find out what the system does only by constructing it; the original ignorance of what the system does is *constitutive of* the system. One invariably starts neither completely knowing nor being completely ignorant about what the system will do – and both knowledge and ignorance are

made visible through the very process of starting anyway. The act of writing specifications for the system gives shape to the system by making some of its unforeseeable consequences visible. In a way, as Derrida would have it, infelicity is constitutive of the possibility of felicity.[30] This is how software constitutes time: because, where does one start *from*? One always starts albeit one does not know what will be. The system one designs now will become the past of the system one will have realized. *Such a system is unforeseeable – but one starts anyway, and this constitutes both the present and the future of the system.*

I want to emphasize here that one must be wary of the idea – suggested by Kinsley and other conference participants - that the closer one is to the stage of 'implementation', that is, of code, the more 'exteriorized', and the less unexpected, the consequences of software become. I want to argue instead that the exteriorization of the software system always brings forth new and different possibilities of unforeseen consequences. To understand this point better, let us look at the concept of 'notation' as it is debated in the Garmisch conference report.

Quite early in the report Perlis points out that 'software systems are mathematical in nature' and that, while a mathematical background is not necessary for a software designer, yet '[such background] can only add to the elegance of the design' (Naur and Randell 1969: 37). Bauer adds: 'Systems should be built in levels and modules, which form a mathematical structure' (37). And Kolence, in his conference paper, clarifies:

> At the abstract level, a concise mathematical notation is required by which to express the essential structures and relationships irrespective of the particular software product being implemented. For example, in the area of computer central processor design, the notation that satisfies this

[30] As I explained in Chapter Two, in his famous re-reading of Austin's theory of the speech acts, Derrida argues that Austin has not 'interrogated infelicity as a law' (Derrida 1988: 15). According to Austin, a 'performative' – such as the wedding formula 'I do' - cannot be true or false, but only felicitous or infelicitous. However, Derrida emphasizes how the possibility of infelicity continues to constitute the structure of a felicitous performative, thus ultimately demonstrating the citational or iterable nature of all language.

> requirements is Boolean Algebra. The notation of Ken Iverson is an attempt to provide an equivalent notation for software.
>
> (Naur and Randell 1969: 38)

In this passage Kolence points out the importance of having what today would be named 'formal notation', or 'formalism' - and what he calls 'mathematical notation' - for design. For him such notation should perform the same functions that Boolean Algebra performs in hardware design, and he quotes Ken Iverson's notation as an example of such a formalism. Most importantly, the concept of notation is associated by Kolence with the idea of 'expression'. I want to argue that 'to express' actually means 'to externalize' here – again in the Stieglerian sense of the exteriorization of consciousness through technology. For this reason, 'notation' can be viewed as one of the possible modes of inscription of software – of which specifications written in natural language, graphical flowcharts and computer programs are all instances. To give but one example, the following sequences of letters and numbers

MOV AX,[202]
A10202
10100001000000100000010

are three equivalent notations for the same state of polarities on a magnetic disk. They are expressed in Assembler language, hexadecimal machine language and binary machine language respectively.[31] They are all part of software, since they are all notations used in software development.

However, in another part of his conference paper Kolence argues for a notation 'which permits an initial description of the internal design of software to be broken apart into successively more detailed levels of design, ultimately ending up with the level of code to be used' (Naur and Randell 1969: 47). Here Kolence attributes to notation a significant role in the sequencing of time. Kolence's short quotation is of immense relevance, since it describes the whole process of software development,

[31] I will develop this point in greater detail in Chapter Five.

and the software system itself, as a process of inscription. I want to argue that 'notation' is here a synonym for the inscription of software and of the process of its development over time. Different kinds of notation allow for different ways of sequencing time.[32] To understand this point better, it is worth noting how in the same paper Kolence argues: '[a] design methodology, above all, should be coherent. ... Software design notation, for example, should decompose naturally from the highest level of design description down to design documents which suffice for maintenance of the final software' (Naur and Randell 1969: 50). This quotation describes the process of software design as a sequencing of notations that coincide with steps in time – that is, it is as a sequenced process of exteriorization. From this point of view, code can well be the most advanced stage of software development, but it does not constitute the complete foreclosure of the unexpected consequences of software.

But in what way does the unexpected keep emerging in code? To understand this point, it is important to examine the so-called principle of extensibility. In the Garmisch conference report extensibility is defined as the possibility of extending the functionalities of the system by reprogramming parts of it (for instance, by adding pieces of software) after the system has been completed and delivered to its users. In his conference paper Letellier argues that a software system should be 'open-ended' – that is, it should have 'syntactic flexibility in the input and modularity in the implementation' (Naur and Randell 1969: 38). More precisely, the capacity of the system to allow for its own extension in the future – or for the broadening of its own life-cycle in the future - is based on the characteristics of the notation in which the system is 'expressed' (as Kolence would have it) – that is, externalized. According to Letellier the system must extend into the future as far as it can and as fast as it can, thus capturing and calculating time, while at the same time leaving a gate to the unforeseeable open.

Importantly, the Garmisch conference report relates extensibility to the concept of modularity. As H. R. Gillette states in his conference paper entitled 'Aids in the

[32] In this sense, computer circuits are themselves inscriptions – or, rather, circuits are notations, in the Stieglerian sense of matter logically organized. I will come back to this important topic in Chapter Five.

Production of Maintenable Software', modularity is one of the main characteristics of a good design – that is, a design that keeps an eye on the future in order to make the system easily maintainable. Gillette writes:

> Three fundamental design concepts are essential to a maintainable system: *modularity*, *specification*, and *generality*. Modularity helps to isolate functional elements of the system. One module may be debugged, improved, or extended with minimal personnel interaction or system discontinuity. As important as modularity is specification. The key to production success of any modular construct is a rigid specification of the interfaces: the specification, as a side benefit, aids in the maintenance task by supplying the documentation necessary to train, understand, and provide maintenance. From this viewpoint, specification should encompass from the innermost primitive functions outward to the generalized function such as a general file management system. Generality is essential to satisfy the requirement for extensibility.
>
> (Naur and Randell 1969: 39 f.)

This passage clarifies that modularity is a way of understanding the system 'functionally' - that is, in terms of tasks performed by smaller parts of the system itself. At the same time, modularity helps in breaking down the process of system production. Since, as we have already seen in the course of this chapter, a module is a self-contained functional unit that requires a minimum of communication with other modules, it can also be easily isolated with minimum disruption of service.[33] Thus, when the software system is finished and in use, modules can be literally 'taken out' of the system and modified (for example, a malfunction can be fixed or a functionality can be added) without stopping the whole system. It can be said that

[33] It must be kept in mind that modules are pieces of software that work together to realize the general purpose of the software system. Such cooperative functioning implies some kind of interaction between modules, and the name given to such interactions is 'communication'. It depends on the nature of the particular software system how such 'communication' is realized. Modules can, for example, exchange signals or share a common area of memory. It is not fundamental now to understand this kind of interaction in detail. Similarly, people developing single modules need to communicate with each other in order to produce modules that are able to work together. Besides, if a module is changed by its programmer, such change may affect the functioning of the whole system. According to Mahoney (2004), the concept of modularization comes from the Fordist model incorporated in the industrialization of software at the end of the 1960s.

modularity introduces a discontinuity in the system - that is, it breaks it down into simpler and more manageable parts - in order to reduce discontinuity in time: namely, to minimize its out-of-service time. In this perspective 'documentation', here understood as the user manuals as well as the technical specifications that help programmers understand the system, is presented as just a part of the broader practice of 'writing' that encompasses the whole of software development. However, the most relevant point here is that, once again, the unexpected consequences of software are inscribed in software itself in all its forms; in particular, they are inscribed in code by means of extensibility and modularity. The combination of extensibility and modularity constitutes a way to calculate the future – but at the same time, since nobody can anticipate what an open-ended system will do, or what can be done with it, it also keeps the possibility of the unforeseeable open.

Before recapitulating the argument of this chapter, let me briefly examine Gillette's use of the term 'documentation'. Indeed, this mention of 'documentation' which refers to those written texts that help the final user understand the software system – generally called 'user manuals' - is extremely relevant, because it is here that the figure of the 'user' makes its appearance in the Garmisch conference report. I want to argue that in the report the 'user' is actually a name given to a part of the process of software design. In the figure of the user both the instability of the instrumental understanding of software and the capacity of software to escape instrumentality through the unexpected consequences it generates become apparent.

In the Garmisch conference report 'the user' makes its appearance as a problematic figure toward which software developers have ambivalent feelings. On the one hand, J. N. P. Hume suggests that designers must not 'over-react' to individual users – that is, in order to develop an effective and usable software system, they must identify the requirements 'common to a majority of users' and focus on them (Naur and Randell 1969: 40). On the other hand, J. D. Babcock argues for the intelligence of the users. He comments: '[t]he users are the people who do our design, once we get started' (40). In doing so, Babcock gives to 'the users' an essential role in the process of software development various decades before the emergence of cooperative Human-Computer Interface (HCI). As I explained in Chapter Two, HCI

technologies aim at facilitating the use of computers by human beings. They presuppose a certain model of the user that has been criticized, for instance, by Matthew Fuller (2003). Fuller highlights the narrowness of the model of the user embedded in HCI, which he understands as 'functionalist' (Fuller 2003: 13). For example, through the human interface of a real-time system a pilot can drop bombs or a stockbroker can move funds efficiently precisely because the interface represents its user in terms of functions performed – that is, in terms of tasks and efficiency. Fuller is critical of such 'idealization' of the user and suggests a shift from the model of the individualised user typical of standard HCI toward different approaches such as Participatory Design – where users provide continuous feedback to programmers in a process of cooperative design. However, according to Fuller the approach of HCI is too much characterized by 'functionalism' to be genuinely critical – that is, however 'human-centred', computer interfaces are designed on the basis of an ideal model of the user. And yet what I want to point out here is that the analysis of the emergence of the 'user' in the Garmisch conference report shows how the possibility of getting important feedback from the users has always been present in the theories and practices of Software Engineering. Even more importantly, the 'user' is inscribed in these theories and practices not just as an 'idealization' or a 'function' of the system, but as a field of forces constitutive of the whole process of software development.

To understand this point better, it is worth looking at the passages of the report where the conference participants express their discomfort in interacting with the user. Manfred Paul depicts the user as someone who 'does not know what he needs', but he couples this with another kind of ignorance: users are actually 'cut off from knowing what is or what might be available' (40). And Perlis adds: 'Almost all users require much less from an operating system than is provided' (40). In these two passages users are understood alternately as unable to understand their own needs – and thus unable to pose clear requests to technology – and overwhelmed by the technological offer – and thus incapable of making the most of the functionalities provided by technology. These complaints about 'users' are a familiar feature not just of Software Engineering, but of the general approach of software developers to their non-technical counterparts (see, for instance, Bolter 1984). However, it would be reductive to interpret such complaints merely in terms

of the difficulties encountered by software practitioners in communicating with non-technical users. Importantly, J. W. Smith notices that designers usually refer to users as 'they', 'them' (Naur and Randell 1969: 40) - a strange breed living 'there in the outer world, knowing nothing, to whom nothing is owed'. He also adds disapprovingly that most designers 'are designing... for their own benefit – they are literally playing games' – they have no conception of validating their design - or at least of evaluating it in the light of potential use (40). This representation of the user as someone 'out there' – someone whose 'needs' should be taken into account in order to validate software instrumentally – is particularly relevant if we want to understand how the figure of the user operates in Software Engineering. I want to suggest that the 'user' and their 'needs' are part of the fictional 'origin' of the software system. As I have shown earlier on, in the Garmisch conference report 'society' is the locus of a projection of the 'demands' that are supposedly made to technology – while at the same time a pre-existing 'problem' is projected in the world 'out there' in order to justify the existence of software. Here I want to emphasize how the figure of the user plays a role analogous to the 'problem' – that is, the user's needs are part of a narrative that software developers construct in order to justify the system they are developing. This is not to say that users do not really exist or that they do not express their demands in terms of what functionalities should be provided by a software system. In fact, the Garmisch conference report takes the communication with users very seriously at all levels. And yet, what I want to point out is that the figure of the 'user' is positioned by the report outside the process of software development in a constant and incomplete movement of 'expulsion' of certain characteristics of software *as* 'user needs'. In Goos's words, software developers need to 'filter the recommendations coming from the outside' (41). A double strategy is at work here, which acknowledges the importance of users while focusing on how to keep them at bay. Randell even laments the amount of time wasted on 'fending off the users' (41). Thus, 'the user' is both constituted and neutralized through various practices of writing: while it is acknowledged that software development is set in motion by the very existence of (potential) users and that it needs their feedback, the very inscription of the software system in terms of specifications, design, code and user manuals acts as a form of containment of the (supposed) user's exigencies.

Even more importantly, as I have suggested above, the figure of the user is associated with the extensibility of software. Not only, according to Letellier, should a software system be 'open-ended' – therefore allowing its developers to modify it in the future (Naur and Randell 1969: 38) – but also, in Gillette's terms, 'documentation' must be provided to users – that is, written texts whose goal is 'to train, understand, and provide maintenance' (39). In Gillette's perspective the inscription of the software system must be done so as to make it easily understandable, and an important part of such an inscription must be written in natural language and delivered to the final users together with code – (the 'working' software system) in order for them to be able to understand and use the system. This part of 'software' – which, as I have explained above, goes under the name of 'user manuals' – is what enables users to engage in an active relationship with software. Ultimately, user manuals allow users to engage with a system whose open-endedness is inscribed in code. Therefore, documentation also constitutes a point where the capacity to take advantage of such open-endedness and to take the system into an unexpected direction is ultimately handed over to the users. This does not mean that any user can actively reprogram any system. In fact, according to the Garmisch conference report, one of the aims of software developers is to make the system 'dumb-proof' – that is, robust and resilient enough to resist 'improper' uses on the part of inexperienced and non-technical users (Naur and Randell 1969: 40). And yet, it seems to me that the figure of the user is the locus where the instrumentality of software is both reasserted by implicitly defining it as a tool to be 'used' and opened up to unexpected consequences. As I will show in Chapter Four, the ambivalent figure of the user will be at the core of many unexpected developments of Software Engineering in the 1980s and 1990s.

To summarize my argument in this chapter, Software Engineering emerges as a strategy for the industrialization of the production of software at the end of the 1960s, where software is understood as a process of material inscription that continuously opens up and reaffirms the boundaries between 'software' itself, 'writing' and 'code'. Software Engineering establishes itself as a discipline through the attempt to control the constitutive fallibility of software-based technology. Such fallibility – that is, the unexpected consequences inherent in software - is dealt with through the organization and linearization of the time of software development.

Furthermore, Software Engineering understands software as the 'solution' to pre-existent 'problems' or 'needs' present in society, therefore advancing an instrumental understanding of software. However, both the linearization of time and the understanding of software as a tool are continuously undone by the unexpected consequences brought about by software – which must be excluded and controlled in order for software to reach a point of stability but which at the same time remain necessary to its development. Through the analysis of the different stages of software development described in the Garmisch conference report I have attempted to demonstrate how the unforeseeable consequences of software are inscribed in software in all its forms; in particular, they are inscribed in code by means of its characteristic of 'extensibility' and 'modularity'. The combination of extensibility and modularity constitutes a way to calculate the future of an open-ended software system – but, since nobody can anticipate what an open-ended system will do, or what can be done with it, it also keeps the possibility of the unforeseeable open. The figure of the 'user' of software represents both the instability of the instrumental understanding of software and the capacity of software to escape instrumentality through the unexpected consequences it generates. In the next chapter I will investigate the development of Software Engineering in the 1970s and 1980s. I will also present the emergence of the open source movement in the 1990s as one of the unforeseen consequences of the Software Engineering of the late 1960s.

4 From the Cathedral to the Bazaar

Software as the Unexpected

> Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected.
>
> (Raymond 2000: 16)

In Chapter Three I argued that in the foundational texts of Software Engineering of the late 1960s 'software' was constituted as a process of material inscription, an act of conceptualisation that took place through the continuous undoing and redoing of the boundaries between 'software' itself, 'writing' and 'code'. In this chapter, I want to investigate how the mutual co-constitution of 'software', 'writing' and 'code' was established in the Software Engineering of the late 1970s and early 1980s. I want to argue that in this period Software Engineering reconfirmed its capacity for continually opening up and reasserting the instrumental conception of software – that is, the understanding of software as a tool. I also want to show how the emergence of open source programming in the 1990s, with its own brand of Software Engineering practices and theories, can be understood as one of the unforeseen consequences generated by the conception of software of the 1970s and 1980s.

Let me briefly recall my argument of the previous chapters in order to show the importance of investigating the relationship between software and instrumentality. In Chapter One I argued that an instrumental understanding of technology is not sufficient to allow us to make sense of new technologies, and especially of software-

based technologies. Drawing on Bernard Stiegler's thought (1998a), I questioned the dominant philosophical conception of technology based on the Aristotelian tradition, which substantially devalues technology as a mere instrument (see Aristotle 1984, 3-4). I then turned to the work of those thinkers who distance themselves from such an instrumental understanding and instead propose a view of technology as constitutive of the human (such as Stiegler himself, as well as Martin Heidegger and Jacques Derrida). Timothy Clark (2000) calls this approach the tradition of 'originary technicity' – a term he borrows from Richard Beardsworth (1996). Thus, I argued for a radical rethinking of the conceptual framework of instrumentality if an understanding of technology is to be made possible.

In Chapter Two I then showed how the study of software as a historically specific technology is important for a radical rethinking of the relationship between technology and the human. Following Derrida's insight on 'new technologies', and his clarification of how difficult it is to conceive them merely in terms of instrumentality (Derrida 1983), I set out to investigate how, on the one hand, software can illuminate the role of technology in the constitution of the human - that is, shed light on 'originary technicity' - while, on the other hand, it also displays a persistent 'complicity' (in Derrida's terms) with the instrumental understanding of technology. I also argued that it is impossible to investigate 'software' in general, since what we call 'software' takes many forms in many different contexts. I focused on the constitution of the term 'software' – in relation with 'writing' and 'code' - in the early theories and practices of Software Engineering at the end of the 1960s. In Chapter Three I showed how Software Engineering was constituted as a discipline in the context of the industrialization of software production and how in Software Engineering instrumentality takes the form of the regulation of the relationship between 'software', 'writing' and 'code'. In particular, I argued that Software Engineering was established as a methodology for the management of the risks implicit in software. In the late 1960s software professionals were faced with the need to evaluate the feasibility of 'large' and innovative software systems, and they understood such feasibility in terms of the management of time. They were aware that software development was not easily manageable, due to the intrinsic fallibility of software, and they instituted Software Engineering precisely as a methodology for calculating the constitutive unpredictability of software. Such

calculation was performed as the sequencing of the process of software development in different spans of activity – in other words, as a sequencing of time. For this reason, I argued that Software Engineering established itself as a discipline through the (impossible) expulsion of the constitutive fallibility of software.

In Chapter Three I also explained how each stage of the process of software development produced, as a result, a piece of writing. Each piece of writing was the point of departure for the following phase, and it was supposed to be used as a tool for developing other written texts. Some of these texts were called 'specifications', some were called 'software' or 'programs' or 'code', but these terms kept shifting. All those texts were considered means to 'make visible' the software system, and each text constituted a refinement in the understanding of the system. In fact, software professionals understood the system *through repeated attempts to build the system itself.* Software professionals developed their understanding of the system through writing – that is, through different forms of material inscription that they named 'specifications', 'documentation' and 'code'. Ultimately, this very process of inscription – what in philosophical terms, and following Stiegler, I called the 'exteriorization' of the system through writing - made the system understandable. I went so far as to suggest that such an inscription was the way in which consciousness constituted itself in relation to software.

By the early 1970s software production was steadily industrialized. Nevertheless, in the 1970s and throughout the 1980s Software Engineering kept questioning itself as a discipline and constantly re-evaluated the validity of its methodologies - especially those deployed for time management. In this chapter, I want to investigate in what way the 'incalculability' of software re-emerged in Software Engineering during the 1970s and 1980s. More precisely, I want to argue that in this second phase of Software Engineering, even though the sequencing of the process of software development appeared consolidated, software kept escaping its instrumentality by giving rise to unforeseen consequences. Not only did the distinction between 'software', 'writing' and 'code' remain unstable, but, most importantly, 'software' remained intrinsically fallible. Furthermore, in the mid-1970s the problem of coordinating different 'minds' (Brooks 1995: 32) – that is, the individual software professionals who took part in a single software project - made its appearance in

Software Engineering. If one views software in terms of the exteriorization of consciousness, it can be said that consciousness was made discrete, and divided into different, and sometimes conflicting, individual consciousnesses. The coordination between individual consciousnesses was understood in terms of 'communication'.

In order to investigate how the problem of software calculability was approached in the 1970s and 1980s, in this chapter I examine two of the fundamental texts on time management in software development, both written by Frederick Brooks in the mid-1970s and mid-1980s respectively. The first one is Brooks' book entitled *The Mythical Man-Month*. Originally published in 1975, it rapidly became the most famous text on time estimates in the whole history of Software Engineering and remained extremely influential for at least two decades. However, ten years later, in 1986, Brooks published an article entitled 'No Silver Bullet', which also became a classic, where he revised his theses of 1975. Such a revision stimulated a heated debate among Software Engineering scholars and practitioners. An accurate analysis of Brooks' work of 1975 and 1986 is key to understanding in what way the calculability of time in software development was conceptualized during the late 1970s throughout the 1980s and early 1990s. In the final section of this chapter I contrast Brooks' understanding of time against the one proposed by the open source movement by looking at the foundational article of Software Engineering for open source programming – namely, Eric Steven Raymond's 'The Cathedral and the Bazaar'. Written in 1997 and republished on-line many times, this article constitutes Raymond's answer to Brooks' original argument on time. As I have already anticipated above, this analysis will lead me to view the open source movement as an unforeseen consequence of Software Engineering that proposes a totally unexpected answer to the problem of coordination between various software designers. It does so by replacing software development as a process of communication with software development as a collective process of overlapping re-inscriptions.

Fred Brooks became the project manager of the Operating System/360 (OS/360) at IBM in 1964. As I explained in Chapter Three, OS/360 contained many technical innovations and became largely popular, albeit it entailed – as it was quite common

at the time – a number of technical flaws.[1] The main problem encountered by the OS/360 project was related to time management. As Brooks would candidly recall ten year later, OS/360 'was late ..., the costs were several times the estimate, and it did not perform very well until several releases after the first' (Brooks 1995: xi). Brooks left IBM in 1965, when he joined the University of North Carolina at Chapel Hill and began a careful investigation of what had actually gone wrong with the OS/360 development in 1964-65. The result was his 1975 book entitled *The Mythical Man-Month*, where he meticulously examined his experience and identified his fundamental mistake as a project manager in terms of wrong time estimates - also putting forward what later on would become known as 'The Brooks' Law' on time management.[2] Let me now examine Brooks' argument in this book in order to investigate in what way he conceptualizes the question of time management in software development.

The imaginative style deployed by Brooks in *The Mythical Man-Month* is replete with metaphors, many of which have a distinct Biblical flavour – first and foremost the famous metaphor of software development as a 'cathedral' which, as I will show later on in this chapter, will be countered by Raymond with his metaphor of the 'bazaar' (Raymond 2000). Let me start here with the examination of the image that opens Brooks' book - namely, the 'tar pit' (Brooks 1995: 4). Brooks writes: 'large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems – few have met goals, schedules, and budgets' (4). What I want to argue here is that the 'tar pit' is nothing but a figure of time – or, more precisely, a figure of the failure of time management in software development. As I showed in Chapter Three, the problem of time management is one of the crucial points debated in the Garmisch conference report – namely, in the main text documenting the emergence of Software Engineering as a discipline during the NATO Conferences of 1968 and

[1] One must be reminded here that the very concept of 'operating system' was highly innovative at the time, and that every new operating system substantially innovated on the previous ones (Glass 2003).
[2] The edition I refer to in this chapter was republished in 1995 as a tribute to the 20th anniversary of *The Mythical Man-Month*'s publication. It comprises a special preface written by Brooks in 1995, the original (unmodified) 1975 book, Brooks' 1986 article 'No Silver Bullet', and a response that Brooks wrote in 1987 to confute many of the critiques directed to his article (plus useful additional material, including a table of the contents of *The Mythical Man-Month* that he still deemed useful in 1995, as well as his remarks on the parts he later rejected).

1969. In order to recall how time management is presented in the report, let me recapitulate my argument briefly here.

According to the Garmisch conference report, the participants in the first conferences on Software Engineering view software technology as a constantly innovating field that advances ever too fast. They perceive the speed of software growth 'with more alarm than pride' (Naur and Randell 1969: 15). The report represents constant technological innovation as a series of big and dangerous leaps forward, and suggests that a step-by-step approach would be a safer way to develop software. At the same time, being confronted with the question of the feasibility of so-called 'large' software systems (of which operating systems such as OS/360 are but one example), the report establishes that the inability to estimate the feasibility of a software project in a reliable way leads to the impossibility of carrying it out step by step, and ultimately to its failure. The failure of a software project is therefore related to failure in time management.

And yet, quite strikingly, Brooks' metaphor of the tar pit seems to suggest that, if there is a problem with software development, it has to do with slowness rather than with excessive speed. No single factor, he adds, seems to be uniquely responsible for the delays of software development: 'any particular paw can be pulled away' from the tar pit, but the accumulation and interaction of simultaneous factors 'brings slower and slower motion' (4). Brooks' image of time as a tar pit is also an image of sinking – or, more precisely, of slower motion: one moves slower and slower, until one sinks. Indeed, sinking seems almost unavoidable, since 'the fiercer the struggle, the more entangling the tar' (4).

Brooks' aim in *The Mythical Man-Month* is to understand the tar pit in which OS/360 'sank' when he was its project manager and, more generally, to comprehend all the tar pits that seem to threaten the development of the majority of large software systems in the early 1970s. It is important to notice that, at the time of his writing, software development is already understood as an industrialized process. The industrialization of software, that - as I showed in Chapter Three - was still under way in the late 1960s, is taken by Brooks as a *fait accompli*. He actually distinguishes 'programming' - the individual task of writing a program - from the

development of a 'programming system product' - the industrial production of large software systems - and clearly establishes that he is interested in the latter. The very expression 'programming system product' emphasises both the systematic and the industrial aspects of the kind of software that Brooks wants to examine. However, the activity of programming is part of the development of software products, and it is from the examination of this activity that Brooks starts. For him, no matter how fulfilling, creative and rewarding, programming also presents many 'woes'. Brooks writes:

> First, one must perform perfectly. The computer resembles the magic of legend in this respect... If one character, one pause, of the incantation is not strictly in proper form, the magic doesn't work. Human beings are not accustomed to be perfect, and few areas of human activity demand it. Adjusting to the requirement for perfection is, I think, the most difficult part of learning to program.
>
> (Brooks 1995: 8)

In this extremely interesting passage Brooks identifies the perfection of the programmer's performance with the perfection of the piece of software he is writing. In fact, it is software that has to perform perfectly – namely, it has to conform to its own specifications which, as we already know, basically describe the required behaviour of the system. Software must be perfect - that is, it must perform as expected: it must be perfectly foreseeable. Intriguingly, Brooks associates such perfection with magic and identifies the good functioning of software with the good functioning of the spell. He also points out that everything is relevant in the written form of a program: a different character or a misplaced space can determine a totally unexpected behaviour of the program when it is executed, thus failing to execute the spell perfectly. In other words, *a misplaced character makes time incalculable.*[3]

[3] It is worth noting how for Brooks a pause in spell chanting equals a space in the text of the program. Although, as I will show in the next chapter, there is no simple equivalence between the use of spacing in a given computer program and the time of its execution, spaces (or blanks) actually determine how a program will be executed. I will propose a detailed analysis of such a relationship in Chapter Five.

In sum, for Brooks perfection corresponds to the perfect control of time – albeit, he adds regretfully, perfection is not of this world. Thus, Brooks seems to acknowledge the intrinsic fallibility of software – fallibility that is also repeatedly mentioned in the Garmisch conference report. However, Brooks maintains that 'adjusting to the requirement for perfection' is a difficult but necessary part of programming, and that the objective of a perfect (enchanted) control of time has to be pursued at all cost. Importantly, here Brooks associates enchantment with the foreseeable, and therefore with instrumentality - in other words, with the perfect performance of software as a tool. His position echoes Franz L. Alt and Morris Rubinoff's interpretation of the 'seductive fascination' of software magic that I briefly examined in Chapter Three (Alt and Rubinoff 1968). Although it might also seem to resonate with Alfred Gell's understanding of the 'enchantment' of technology, Brooks' conception of enchantment is much closer to Alt and Rubinoff's argument than to Gell's.[4] For Gell, 'enchantment' indicates the mystifying social effects of technology: when the functioning of technology is not understood by society, the latter misrepresents the former as the result of magic. Even though Alt and Rubinoff boast about the esoteric aspects of technology, for them the 'seductive fascination of software' is first of all related to intellectual pleasure and to the mastery of the human 'wizard' over technology. Similarly, for Brooks 'magic' does not obscure the functioning of technology – it does not mean 'mystery'. In fact, magic sheds light on software, it 'demystifies' it: since the secret of a well-functioning spell is its perfect formulation, all we need in order to make software work is to write it perfectly. For this reason, magic is perfectly predictable, and so must be software: a good spell does what it is supposed to do, as much as a good software system meets its specifications.[5]

Even more importantly, and consistently with his metaphor of the tar pit, Brooks downplays the importance of the speed of software growth that just a few years earlier in Garmisch was considered such a big problem (Naur and Randell 1969: 14). Brooks points out that, although software systems start becoming obsolete as

[4] One must be reminded here how Alfred Gell (1992) establishes a strong relationship between art, technology and magic. He views art as a special form of technology and argues that the magical prowess, which is supposed to have entered the making of the art object, depends on the level of cultural understanding that surrounds it. In fact, society misrepresents to itself the technically achieved excellence of a work of art as a product of magic.

[5] In the second section of this chapter I will argue that Raymond (2000), in turn, associates magic with the unforeseeable: for him, the emergence of working large software systems from the practices of open source programming is a sort of 'magic' – or even a 'miracle'.

soon as they are completed, newer products take time to be designed and implemented. Therefore, the obsolescence of an existing software product is not as fast as it might seem. 'Of course', Brooks writes, 'the technological base on which one builds is *always* advancing. As soon as one freezes a design, it becomes obsolete in terms of its concepts. But implementation of real products demands phasing and quantizing' (Brooks 1995: 5 [original emphasis]). For Brooks, being time-consuming, the very activity of developing a 'real' software system imposes limits on the speed of technology. Although technology remains a process of constant innovation, once again Brooks seems to be concerned with the slowness of software growth, rather than with its speed. For him, software is anything but 'a device that goes faster than its own time' – the image proposed by Stiegler to indicate the new relationship that contemporary technology entertains with time (Stiegler, 1998a: 15).[6] But since software was still undergoing a process of constant innovation in the mid-1970s, in what way could software professionals perceive software as 'slow'?

Crucially, Brooks understands slowness first and foremost as the fact that projects tend to fall behind schedule. He clarifies: 'more software projects have gone awry for lack of calendar time than for all other causes combined' (14). Brooks' 'calendar time' stands for industrialized, discrete, 'quantized' time. From a Heideggerian perspective, this is 'enframed' time – namely, time as a resource of which programmers might fall short during the process of software development. Thus, for Brooks software development is worryingly 'slow' in terms of industrial time - or the sequenced time that is already common practice in the industrial production of software in the mid-1970s. After all, one needs a deadline in order to be late. For Brooks the slowness of software is not a general slowness in the technological advancement of software; rather, it is software's resistance to conform to the deadlines of industrial production.

[6] Stiegler's favoured analogy is that of 'a supersonic device, quicker than its own sound', whose breaking of the sound barrier provokes 'a violent sonic boom, a sound shock' (Stiegler, 1998: 15). In Chapter Three I showed how in the late 1960s software was considered a technology that continually exceeded its own boundaries through constant innovation. To name this process, software professionals coined the term 'software crisis' (Gries 1989: 98).

In sum, Brooks attributes software's 'slowness' – the 'tar pit' - to the ineffective management of time, and particularly to bad time estimates. And yet, he upholds the perfect management of time as the (impossible) ideal of software development. Brooks debates here the fundamental problem of the industrial production of software – namely, the establishment of realistic time estimates, which in turns leads to the setting of reasonable schedules, so that the corresponding deadlines can be met. In large-scale software systems, he notices, a project manager and his staff are in charge of these decisions, and they make them on the basis of their experience as programmers and possibly as project managers of earlier projects. And yet, Brooks warns, 'all programmers are optimists' (14). Programmers – here describing all the technical personnel involved in a software project, including project managers - always assume that there will be 'enough time'. In fact, this notion of 'optimism' is crucial for the understanding of Brooks' concept of software's slowness. What I want to argue here is that 'optimism' actually conveys the *impossibility of estimating the unexpected.* Programmers are incapable of calculating all the possible consequences of software development, including those that will require more labour than expected and that will therefore delay the completion of a project. As I pointed out in Chapter Three, the Garmisch conference report explicitly relates the 'software crisis' of the late 1960s to the 'craft mentality' of software professionals, and presents the 'scientific planning' of time, as well as software industrialization, as the solution (Galler 1989: 97; Naur and Randell 1969: 17). Seven years later, Brooks' book shows that the question of time management is still unresolved, although it is represented in quite a different way – namely, through the depiction of software as something that is advancing slower than expected. However, the production of time estimates is no easy task, and it is directly linked to the sequencing of the process of software development. Let me now investigate this connection further.

As we have seen, the foundational texts of Software Engineering are quite clear about the difficulty of following 'a sure path' in software development – in other words, of conforming to a step-by-step sequence in which a system is completely and exhaustively specified before it can be realized (Naur and Randell 1969: 32). The Garmisch conference report presents a rather clear view of such a 'paradoxical' aspect of Software Engineering. For instance, it acknowledges that the

specifications of a software system are never complete, and that inconsistencies become clear only later on, when the process of implementation starts – i.e., inconsistencies are made visible *by implementation*, and this requires an iterative improvement (or a rewriting) of specifications. In fact, the sequencing in time of the process of software development, which is assumed as the scientific basis of Software Engineering, is an impossible task. One invariably starts to do what one wants to do before knowing what it is. The paradox is that a software project is successful when it meets its specifications - that is, when it does what its specifications say it does - but the specifications were written when one did *not* actually *know* what the system was supposed to do. Thus, in order to clarify Brooks' idea of the 'slowness' of software, it is important to ask the following question: in what way does Brooks deal with the problem of the sequencing of the process of software development and with the emergence of inconsistencies during such a process? To understand why for Brooks software is slow (or late), it is necessary to find out in what way he sets deadlines to software development.

In order to answer the above question, it is worth noting that Brooks understands software development as a 'creative' process.[7] In *The Mythical Man-Month* Brooks explicitly quotes Dorothy Sayers' book, *The Mind of the Maker,* which explores at length the analogy between human creation (especially literary creation) and the Christian doctrine of the Trinity. Sayers divides any creative activity into the three stages of the 'idea', the 'implementation' and the 'interaction' – a division that will remain fundamental in Brooks' thought and that ten years later he will still support, by explicitly relating it to the Aristotelian separation between the essential and the accidental, the ideal and the material. It is worth noting here that what Brooks refers to as 'Aristotelian' is in fact the distinction between the ideal and the material established by Plato in the *Republic* (and then reflected in Aristotle's philosophy). Such a distinction is expressed in the well-known myth of the cave (*Republic*, 514a-519a), in which Plato depicts ordinary human beings, deprived of philosophical education, as prisoners in a cave forced to look at shadows rather than at real things

[7] The Garmisch conference report also gives a relevant role to 'creativity' in software development: it is an unforeseeable creative leap on the part of programmers that allows the transition between the different stages of software development – particularly, the transition from a supposedly pre-existent problem (to which software constitutes the solution) toward software itself. As I will clarify in a moment, Brooks proposes a different understanding of 'creativity': for him, creativity constitutes the foundation of the whole process of sequencing in software development.

– by this implying that the ordinary world of sensible objects is far less real than the world of concepts or Forms (Plato 2000).[8] However, in his 1975 book, Brooks paraphrases Sayer as follows:

> A book... or a computer, or a program comes into existence first as an ideal construct ... It is realized in time and space, by pen, ink, and paper, or by wire, silicon, and ferrite. The creation is complete when someone reads the book, uses the computer, or runs the program, thereby interacting with the mind of the maker.
>
> (Brooks 1995: 15)

This passage clarifies that for Brooks the three stages of creation roughly coincide with the conception of a work of art or technology, its concretization in a material form, and its fruition on the part of viewers, readers and so on. But then Brooks adds, somewhat unexpectedly: 'for the human makers of things, the incompleteness and inconsistencies of our ideas become clear only during implementation. Thus it is that writing, experimentation, "working out" are essential disciplines for the theoretician' (15). Notwithstanding his belief in the separation between the conception of a project and its realization, or between the conceptual and the material dimensions – which will become even clearer in his article of twenty years later, where he will introduce an even less tenable distinction between 'concept' and 'representation' – here Brooks seems to be aware that the process of software development is essentially one of 'exteriorization'. In turn, such exteriorization involves the iteration between different stages.

To understand this point better, we should note how Brooks' distinction between conception and realization mirrors the one between software as a product and software as a process that is also constantly opened up and reaffirmed in the Garmisch conference report. As I showed in Chapter Three, a 'point of opacity' of the early texts of Software Engineering is the distinction between process and product, which, albeit necessary, cannot be maintained at all times. In fact, the Garmisch conference report makes clear that a certain confusion exists between the

[8] Brooks does not reference Aristotle's work directly. However, the distinction between essence and accident is formulated by Aristotle in *Metaphysics* VII 4-6, 10-11 and 17 (Aristotle 1984).

concept of the software system and the concept of its development. While the software system is envisaged as a set of interrelated components that work together in order to achieve a goal, its development is described as the process through which the system itself is constructed (or 'written'). Nevertheless, the system and its development are never clearly separable. For instance, the failures in the process of software development – failures that lead to time and investment spinning out of control – are attributed to the poor understanding of the software system in the first place. On the other hand, understanding the system is presented as a matter of 'making it visible': for instance, 'code', or the implementation of the software system, can make visible inconsistencies or errors in a flow chart - that is, in the specifications of the system. For this reason, some iteration between the stages of specification and implementation is necessary. In Chapter Three I also argued that if the person going through the iteration cycles is one and the same, then their understanding of the system (what in philosophical terms we might call their 'consciousness') develops *through* the inscription of marks (the flowcharts, the code) – in other words, the very process of the exteriorization of the system through writing makes the system understandable.

As I have noticed above, to some extent Brooks acknowledges that the process of the 'exteriorization' of the system makes its inconsistencies visible. Indeed, he seems to concede that software exists *only* as exteriorization. The distinction between 'the ideal' and its 'realization' seems therefore to become undone at the very moment of its establishment. In fact, Brooks appears quite unsure as to how to keep the two aspects apart. I want to point out how his argument encounters a major difficulty – or better, a point of opacity - here. On the one hand, he states that the 'realization' of the system takes time because of the difficulty of manipulating the 'physical media' involved, while on the other hand he attributes such a difficulty to inconsistencies that can be found in the very conception of the system. From this second point of view, difficulties would ultimately reside at the 'ideal' level, and the 'realization' of the system would actually coincide with its conceptual clarification. Brooks introduces further ambiguity into his argument by surprisingly claiming that the medium of programming is 'an exceedingly tractable medium'. He explains:

> [programmers] build from pure thought-stuff: concepts are very flexible representations thereof. Because the medium is tractable, we expect few difficulties in implementation; hence our pervasive optimism. Because our ideas are faulty, we have bugs; hence our optimism is unjustified.
>
> (Brooks 1995: 15)

In this passage, Brooks suggests that programmers' optimism - that is, their belief in the calculability of time – relies on their faith in the tractability of the whole system (both as a concept and as its realization) and that, in so doing, they overlook the system's inconsistencies. In sum, Brooks places the difficulty of developing a software system alternatively at the 'material' level, albeit this is contradicted by the supposed tractability of the computer as a medium, and at the 'ideal' level, therefore making 'realization' (or 'implementation') the place where difficulties are actually clarified. This ambivalence suggests that the separation between conception and realization is ultimately untenable.[9]

To recapitulate the above argument, although the distinction between the conception of a software system and its material realization (or implementation) appears quite unstable, Brooks invokes precisely the Platonic distinction between the ideal and the material in order to provide a foundation for his model of the sequencing of time in a software project. At this point, the crucial problem of Brooks' theory of time management emerges – namely, the 'mythical man-month'.

To expand on this point, it is worth noting how, according to Brooks, the development of a large system 'consists of many tasks, some chained end-to-end', and that when one considers this, '[t]he probability that each will go well becomes vanishingly small' (16). Importantly, for Brooks software development is a chain of

[9] Such a distinction is already quite unstable in the Garmisch conference report (Naur and Randell 1969: 30). Furthermore, as I argued in Chapter Two, the very distinction between the material and the symbolic is questionable in itself. Drawing on Jacques Derrida's problematization of Western thought, I established how materiality is implicit in every sign, and how the process of signification itself must be understood as material. In the tradition of Western metaphysics, the signifier (for instance a sound or a graphic sign) is a material entity, while the signified belongs to the realm of concepts. For Derrida this opposition is the foundation of all the other oppositions that characterize Western metaphysics, including the one between the 'ideal' and the 'material'. I will return to this important point later on in this chapter.

tasks – or, in André Leroi-Gourhan's words, an 'operating sequence'.[10] Moreover, and for the first time, Brooks calls the attention of the reader to the fact that the time of software development involves waiting. The fact that some tasks are 'chained end-to-end' means that one has to wait for the conclusion of a certain task before beginning the next one. According to Brooks, poor estimates - that is, optimistic estimates that set impossible deadlines, which in turn will not be met and will result in the lateness of the project - hide the mistaken 'assumption that men and months are interchangeable' (14). At this point, the well-known unit of measure of the 'man-month' is expressly questioned by Brooks.

As I mentioned in Chapter Three, the man-month and the man-year were already in use in the 1960s to measure the size of software systems and the duration of software projects - and they are still popular today. They are defined respectively as the number of months or years that an average programmer would spend on the system if he or she were to develop that system by themselves. For a number of very large software projects the unit of man-millennia can also be used. Since the difference between these units is not immediately relevant to my argument, I want to propose here the comprehensive name of man-time units for them.[11] However, Brooks argues that the man-month is a fallacious unit of measure that leads to erroneous time estimates. The cost of a software project does vary 'as the product of the number of men and the number of months', Brooks argues, 'but progress does not' (16). Hence, the man-month as a unit for estimating the duration of a software project is a 'dangerous and deceptive myth', because '[i]t implies that men and months are interchangeable' (16). I want to highlight here that Brooks does not contest the 'enframing' of the human as a resource in industrialization, to use a Heideggerian expression again. In principle, for him the human *is* a commodity.

[10] As I explained in Chapter One, the concept of operating sequence (*chaine operatoire*) is generally considered to be Leroi-Gourhan's fundamental contribution to the disciplines of anthropology and archaeology. He defines the 'operating sequence' as the sequential organization that underlies both language and technology (Leroi-Gourhan 1993: 114). He also relates the emergence of alphabetic writing to the process of the sequentialization of symbols, which he names 'linearization'. In *Of Grammatology* Derrida (1978) expands on Leroi-Gourhan's theory and relates the emergence of phonetic writing to the linear understanding of time and history. In Chapter Three I emphasized that the linearization of the time of software development was one of the primary objectives of the NATO conferences on Software Engineering, although the participants were also aware that the process of software development was not completely linearizable. As I will show in a moment, for Brooks the incomplete linearization of time constitutes the limit of software productivity.

[11] Brooks' expression 'man-month' has obvious sexist undertones. Later on, Raymond will propose his own unit of measure, the more politically correct 'person-hour' (Raymond 2000).

What he questions is just the functional equivalence of the human and time as resources *in all parts of software development.* He claims: '[m]en and months are interchangeable commodities only when a task can be partitioned among many workers *with no communication among them*' (16 [original emphasis]). This absence of communication is for Brooks a characteristic of activities such as 'reaping wheat or picking cotton' (16), but not of software development. Here Brooks is not merely contrasting the manual repetitiveness of these rural tasks with the creativity of software development. Rather, what he has in mind is the fact that these activities *do not involve communication.* Brook's reapers do not need to communicate with each other. What Brooks is actually trying to demonstrate is that, albeit time must be quantized, or made discrete, to manage software development, man-time can be useless in certain circumstances - namely, those circumstances that involve communication. In fact, I want to argue that the fallaciousness of Brooks' man-month is the image of the always imperfect linearization of the time of software development.

In order to understand this point better, let me now examine how the two main problems identified by Brooks – namely, waiting and communication – impede the sequencing of time, and therefore make time estimates unreliable. As for waiting, according to Brooks, '[w]hen a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on the schedule' (17). What Brooks means here is that, if the sequencing of software development includes waiting, then the chained tasks cannot be parallelized – they cannot be undertaken by two different groups of programmers at the same time. If a project involves non-parallelizable tasks, its duration cannot be calculated in man-time, because adding more workforce is useless when tasks cannot be performed independently. As for communication, Brooks explains that 'in tasks that can be partitioned but which require communication among the subtasks, the effort of communication must be added to the amount of work to be done' (17). Here again man-time cannot be applied because, albeit tasks are parallelizable, if they require communication more man-time is needed to carry out communication itself. Brooks writes: '[s]ince software construction is inherently a system effort – an exercise in complex interrelationship – communication effort is great, and it quickly dominates the decrease in individual task time brought about by partitioning. Adding more men

then lengthens, not shortens, the schedule' (19). In sum, for Brooks both waiting and communication make man-time an unreliable unit of measure for time estimates and are obstacles to the linearization and calculation of time.

Although insisting on the separation of tasks in order to make the project manageable, Brooks acknowledges that such a separation can never be complete. In fact, he states that when a software project is behind schedule, one adds manpower. And yet, since human beings and time are not interchangeable, 'by adding men, you do not add time' (21).[12] According to Brooks, after adding manpower a project almost always ends up being later than ever. Thus, he proposes 'Brooks' Law': 'Adding manpower to a late software project makes it later' (25). He calls it 'the demythologizing of the man-month' and explains:

> The number of months of a project depends upon its sequential constraints. The maximum number of men depends upon the number of independent subtasks. From these two quantities one can derive schedules using fewer men and more months. (The only risk is product obsolescence). One cannot, however, get workable schedules using more men and fewer months.
>
> (Brooks 1995: 25 f.)

In this passage, Brooks argues that the discreteness of the software system - that is, the number of parallel tasks needed for its completion – is the basis for calculating the maximum number of programmers that can work together on the system. However, the sequential constraints of the tasks - in other words, waiting - determine the minimum number of months that the process of development will take, *no matter how many programmers will work on it*. Moreover, an indefinite amount of effort and time will go into communication, making the unit of man-time even less reliable as the basis for time estimates. This is turn hinders the

[12] The fact that the new programmers need training must also be taken into account. Brooks gives the name of 'regenerative effect' to the time spent on training the newly hired programmers, on repartitioning the project ('time rescheduling') and on communicating in order to coordinate these tasks (Brooks 1975: 25).

linearization of the time of software development – that is, it makes time less and less predictable.

Furthermore, as I explained in Chapter Three, one of the main problems of software development consists in the overlapping of the different stages of the same project and in the iteration between them (for instance, implementation usually begins before specifications are completed, code in turn sheds light on specifications' inconsistencies, which eventually leads to specifications being rewritten). The necessity of iteration is partially acknowledged by Brooks. And yet, it is precisely iteration that calls into question Brooks' conception of 'waiting', and in general of partitioning tasks and of sequencing time. In fact, in order to wait for a certain task to be completed, one must have faith in the possibility of such completion – but the iteration inherent in software development always makes completion uncertain.[13]

To summarize the above argument, Brooks' 'demystification of the man-month' amounts to the demonstration that the time of software development is never completely calculable. And yet, such calculation – that is, the linearization of time and the attempt to anticipate the duration and outcome of each stage of software development – is necessary in order for software to exist. In Brooks' terms, a perfectly calculable and foreseeable software project would include a finite number of parallel tasks which do not overlap, without iteration and without communication among programmers. According to Brooks' himself, such a project cannot exist. Nevertheless, Brooks' strategy for coping with the actual paradoxes of software development involves maintaining up the separation between tasks (based on the

[13] The fact that subtasks are actually completed - and that software system are actually finished and delivered to users – is an effect of time management: at a certain point in time, a system is declared stable (and marketable) and a 'release' is consolidated. As I explained in Chapter Three, a release is version of the system that is deemed complete and correct enough to be delivered to its users. Such a delivery does not prevent system developers from improving and extending the system until they reach a new (and supposedly better) stable version of it. Later on in this chapter I will show open source's different treatment of releases: while the Fordist sequencing of time implicit in Brooks' time management aims at keeping the number of public releases of a system as low as possible (in order to present users with fewer but stable versions of the system), open source software is delivered as often as possible, and users themselves are responsible for debugging it.

Platonic distinction between the ideal and the material) and controlling the communication and coordination between them.[14]

In order to understand Brooks' strategy better, let me now investigate his model for the organization of the software project. This model is based on the concept of the 'surgical team', which Brooks derives from Harlan Mills (1971) and which establishes a neat separation between the 'fewer minds' and the 'many hands' involved in any software project (Brooks 1995: 32). Brooks argues that a software system must be developed at the conceptual level by just one or at most two professionals (the 'minds'), and that other manpower (the 'hands') must be brought in only at the later stage of implementation. The separation between the ideal and the material is now explicitly connected with the number of participants in the software project. In Brooks' model, 'the system is the product of one mind – or at most two, acting *uno animo*' (35). This sounds very much like Brooks' own dream of the elimination of any need for communication, at least at the 'ideal' level of the project. Furthermore, if we view software in terms of the exteriorization of consciousness, it can be said that here consciousness is made plural, and divided into different, and sometimes conflicting, consciousnesses. In fact, never before have we come across a specific discussion of *whose* consciousness is exteriorized in software. It is Brooks who comes up with the issue of the number of 'minds' at work on a software project. For him, the coordination of the process of software development is not just a matter of making time discrete, nor of making software visible as inscription, but of coordinating a plurality of consciousnesses. For Brooks, if software is exteriorized consciousness, perfect software is the exteriorization of *only one* consciousness.

For the software system to have conceptual integrity, Brooks argues, hierarchy is necessary, as well as 'a sharp distinction ... between architecture and implementation', while 'the system architect must confine himself scrupulously to architecture' (37). This last passage means that implementers (the 'hands') must

[14] As I will show later on in this chapter, Raymond's strategy will actually downplay the separation between tasks and minimize the control over task coordination – nonetheless maintaining a discrete and linear conception of time.

work only on coding without affecting system specifications.[15] At this point, Brooks' famous metaphor of the cathedral enters the picture. Contrary to other European cathedrals, which bear the traces of the many different generations that have built their different parts – and, Brooks adds, that have not resisted the temptation to 'improve' upon the designs of the earlier ones - the Reims cathedral shows an extraordinary architectural unit, a 'glorious' coherence that Brooks praises emphatically (42). '[M]ost programming systems', he muses, 'reflect conceptual disunity far worse than that of cathedrals', albeit they did not take centuries to build (42) – and yet, such disunity does not arise from 'a serial succession of master designers, but from the separation of design into many tasks done by many men' (42). Brooks' main point here is that 'conceptual integrity is *the* most important consideration in system design' (42) – and he will still contend this twenty years later. A software system needs to 'reflect one set of design ideas' (42) and to 'proceed from one mind', or from a very small number of 'agreeing resonant minds' (44). On the other hand, the pressures of schedule - namely, of the calculated time of software development - require many 'hands', and therefore a division of labour, that must be managed through the clear separation between 'architecture' and implementation. 'Architecture', which here, for the sake of simplicity, I take as another name for 'specifications', is defined by Brooks as 'the complete and detailed specification of the user interface' (45). Importantly, for Brooks specifications define software as a tool by establishing what the user can do with it. I want to emphasize here how the specifications of the system constitute the definition of the system's *instrumentality*. In terms of the 'what' and the 'how', as we already know, the specifications are the 'what' of the system, while implementation (or the 'realization' of the system) is the 'how'. Brooks illustrates this with the example of the clock, where the specifications correspond to the face, the hands, and the winding knob, from which anyone (even a child) can tell time once they have understood how to read them; the implementation is 'what goes on

[15] Brooks here is reasserting the separation between the 'what' and the 'how' – specifications and implementation - that I examined in Chapter Three. Simply put, architecture describes 'what' a system is supposed to do, while implementation establishes 'how' it must do it. In Chapter Three I also showed that this separation is unstable, since the iteration between code and specifications leads to the 'how' affecting the 'what'.

inside the case'.[16] However, the relevant question for Brooks is this: does the separation between specifications and implementation set in place a kind of 'aristocracy of the intellect' (versus a more democratic instance of the model of software development – say, a model in which any idea, either suggested by the architects or the implementers, is taken into account and incorporated in the system specifications)? It is worth noting at this point that the idea of open source as a more democratic model of software development is widely circulated in contemporary media studies (see, for example, Fuller 2003). Open source can be thought of as a non-hierarchical way of developing software. It actually thrives on the incorporation of 'any' idea - coming from any programmer involved in any stage of the process of software development into one version or another of a software product. And yet, I want to emphasize that open source is less concerned with the principle of introducing more democracy into software development than with the technological interest in getting things done. As I will argue later on in this chapter, open source relies on a different sequencing of time, but its conception of software is still characterized both by an instrumental understanding of software as well as by the tendency of software to escape its own instrumentality. However, Brooks rejects the option of incorporating good ideas into the system specifications coming from all the team including the implementers, because for him this would be disruptive of the conceptual coherence of the system. Instead, he recommends that ideas that do not fit in the original conceptual integrity of the specifications be left out. For Brooks, the governing principle of integrity is exclusion. This is 'an aristocracy that needs no apology' (46), since it is functional to the control of the time of software development.

Nevertheless, in order to mitigate the previous part of his argument, Brooks adds that implementation is just as creative as specifications writing after all. This surprising point needs further explanation. Indeed, he claims that 'discipline is good for art' (46) and that the imposition of a limit – consisting of the consolidated specifications of the system - on implementers actually nurtures their search for creative solutions. As long as implementers focus only on the 'how', without ever

[16] That Brooks chooses here the very image of the calculation of time as an example is very appropriate, since instrumentality and the sequencing of time are tightly correlated.

questioning the 'what', they have the greatest freedom: they are artists that need to be subjected to some kind of containment in order to thrive, but such containment here is not identified with the physical limitations of the computer on which they are working (as one might expect, given Brooks' separation between ideal and material, content and medium), but with the temporal limit established by the division of labour. Thus, the inscription of the software system constitutes *and* limits the freedom of inscription of the implementers. Even more interestingly, while arguing for the creativity - or the relative freedom - of implementers, Brooks presents a more flexible solution to the problem of time sequencing than the one he proposed earlier on in his book. Now he remarks that implementers do not need to wait for the specifications to be completed before starting their job. In fact, as soon as the first draft of the specifications becomes available, they can start implementing 'some functions' and writing 'some code'. Thus, once again, the iteration typical of software development creeps back into Brooks' model. As soon as he allows for an overlap between the activities of specification and coding, he is forced to admit that implementers can make visible some inconsistencies in the specifications of the system.

Moreover, such an iteration is possible precisely because code is a re-inscription of software – that is, a different exteriorization of consciousness. Code cannot possibly be the mere result of the activity of the 'hands', since, from the perspective of originary technicity, the very act of coding is constitutive of the programmers' consciousness. I want to argue that, by separating specification from coding, Brooks attempts to perform the impossible expulsion of the 'hand' from the 'mind' – or the separation between hand and mind, which is precisely what the tradition of originary technicity shows as untenable. Furthermore, as I showed in Chapter One, consciousness is never one and the same, perfectly transparent to itself – contrary to

what Brooks seems to imply with his dream of software as conceived by 'one mind'. But consciousness is always already at odds with itself in time (Stiegler 2003b).[17]

Ultimately, for Brooks software is a dream of absolute transcendence and transparency. As we have seen above, Brooks roots his theory of time management in the separation between the ideal and the material. 'Good' software must be conceived by a unitary consciousness in perfectly consistent terms. Subsequently, it can be transparently 'communicated' to software developers who proceed to embody it into a material form. Again, such perfect communication is based on transcendence. Consistently with the Platonic tradition, Brooks understands software as the transcendental signified that can be communicated by an empirical signifier. Transcendentally conceived by one mind and transparently communicated to the 'hands', software is then ready to be materialized – that is, implemented – into a 'real' system. However, and once again, from the point of view of originary technicity, such a distinction between the transcendental and the empirical is problematic. In Chapter Two, drawing on Derrida (1978), I agued for the materiality of *every sign*. One must be reminded here how for Derrida 'writing' takes precedence over orality not because writing historically existed before language, but because we must have a sense of the permanence of a linguistic mark in order to recognise it and to identify it. Ultimately, the sense of writing is necessary for signification to take place. But we can have the sense of the permanence of a mark only if we have the sense of its inscription, of its being embodied in a material surface. In other words, language itself is material; it needs materiality (or rather, it needs the possibility of 'inscription') to function *as* language. If materiality is the condition of signification, then every code is material. Even more precisely, the possibility of inscription is the condition of code's functioning. Software can function only through materiality - not because it has to run on a processor, nor

[17] As Stiegler observes 'that which makes consciousness be self-consciousness (i.e. consciousness that is conscious of contradiction with itself) is the fact that consciousness is capable of externalising itself' (2003b: 163). One writes 'it is dark', and when one rereads the note twelve hours later it is light. This produces, as Stiegler himself further clarifies, 'a contradiction between times', namely the time of consciousness when one wrote this and the time of consciousness when one reads this. Yet, one has the same consciousness, which is therefore 'put in crisis' (Stiegler 2003b: 163), and this crisis in turn raises self-awareness. The act of inscription - that is, of exteriorization - ultimately constitutes interiority, which does not precede exteriority or vice versa. As I have explained earlier, for Stiegler (again drawing on Leroi-Gourhan) the process of exteriorization constitutes the foundation of temporality, of language and of technical production.

because there are economic forces behind it, but because it is (also) code, and code functions only through materiality, because materiality is what constitutes signs (an therefore codes).

To recap the above argument, for Brooks perfect software is perfectly predictable, and such perfect predictability (or calculation of time) is worked out at the (pure) conceptual level and transmitted (as a pure, invariable content) to a workforce that will embody it into some material substrate. This transmission is a perfect process of communication through non-interfering media. Unfortunately, this perfection has been declared from the beginning as a non-worldly goal. Occasionally, mis-communication can occur: for instance, Brooks establishes a relationship between errors in the software system ('bugs') and mis-communication between 'minds': for him, bugs are the result of miscommunication. But a process of communication that can fail is not perfectly transcendental and transparent. As I have argued above, there is actually no pure content of communication that would be separable from its material substrate. Thus, Brooks' dream of software as the brain-child of one-ness reveals itself as an impossible dream.

However, Brooks still attempts to maintain the fragile distinction between specification and implementation through an accurate analysis of the different texts produced in the various stages of software development. To understand this point better, let me now examine these texts, and particularly the one he considers the most important – namely, the 'user manual'. Brooks defines the user manual - or specifications - as the main product of the architect's labour and as a 'necessary tool' (62). It is important to notice that this written text is explicitly defined as a tool. Therefore, writing is clearly positioned as instrumental to the development of the software system. Moreover, given the process of iteration that goes on between specification and implementation, Brooks adds: '[r]ound and round goes [the specifications'] preparation cycle, as feedback from users and implementers shows where the design is awkward to use or build' (62). Brooks insists that changes in the specifications 'be quantized' through 'dated versions appearing on a schedule' (62). Here Brooks starts calling the specifications - or the user manual - 'the book', a term that deserves a careful analysis. In fact, 'the book' must have different and clearly separated versions according to the changes introduced in the software system

during its development. For Brooks the history of the different versions of 'the book' is the history of the system. The discretization (or 'quantization') of time here coincides with the consolidated versions of the book, and again writing constitutes the linearization of the time of software development. The book must describe the interface in a 'complete' – exhaustive – way; it must define the 'what'. However, it needs to 'refrain from describing what the user does not see' – namely, the 'how', which is the implementer's business (62). By establishing all this, Brooks institutes 'the book' as the foundation of the history of the software system – a poignantly Biblical image.

Even more importantly, and consistently with the state of the art of memory supports in the 1970s, the book is essentially paper-based. Brooks goes to great lengths to suggest an economical way to introduce changes into the thick and complex 'book'. For instance, he suggests using a loose-leaf folder and retyping just the pages that need to be changed. He also recalls how during the OS/360 project a transition was attempted from a paper-based manual to one based on microfiche technology, and laments that, although the microfiche is much smaller, the user manual remains more manageable *as it can be annotated on the margins*. Brooks' position might seem quite backward-looking from the contemporary point of view of ever-changing, faster and smaller digital memory supports. But one must recall here that professionals often take quite a conservative approach to new technologies (Bolter 1984). Innovation is pursued in terms of software development, but technologies that are considered instrumental to such development (for instance, memory supports for specifications) are treated conservatively and innovations are not easily introduced. One must also be reminded here of Derrida's reflections on the fundamental non-instrumentality of such 'tools' – for instance, of his analysis of the influence of the technologies of communication on the field of psychoanalysis in *Archive Fever* (Derrida 1996). In this work, Derrida argues that contemporary changes in technologies of communication fundamentally affect our structures of thought, and that 'what is no longer archived in the same way is no longer lived in the same way' (Derrida 1996: 18). For him, 'the technical structure of the *archiving* archive also determines the structure of the *archivable* content even in its very coming into existence and in its relationship to the future' (17). In *Archive Fever* Derrida also reflects on how far the field of psychoanalysis may have been

influenced by the technologies of communication to which Freud and his disciples had access (such as print media and postal services), and whether it would have been different if they had access to contemporary technologies of communications such as e-mail (17). What I want to argue at this point is that Brooks' rigid sequencing of time - which is typical of the Software Engineering of the 1970s - is not separable from the difficulty of erasing and rewriting a written page.

Thus, paper-based technology shapes the separation between architecture and implementation, because it makes it quite difficult to re-inscribe the system when a previous decision is changed. Simply put, *paper shapes software.* Brooks is concerned with maintaining the boundaries between different stages of software development and with containing changes, because the not-so-flexible technology of inscription cannot handle too many re-inscriptions of the system (and especially conflicting ones). Furthermore, Brooks oscillates between the conception of the book as the recording of the history of the software system and as the realization of the system itself. He actually states that the book 'is not so much a separate document as it is a structure imposed on the documents that the project will be producing anyway' (75) – therefore, I want to add, on the system. He writes:

> technical prose is almost immortal. If one examines the genealogy of a customer manual for a piece of hardware or software, one can trace not only the ideas, but also many of the very sentences and paragraphs back to the first memoranda proposing the product or explaining the first design. For the technical writer, the paste-pot is as mighty as the pen.
>
> (Brooks 1995: 75)

It is extremely important to notice here that Brooks understands 'the book' as an archive - namely, as the memory of the changes undergone by the system during its development. From this point of view, in Stiegler's terms, 'the book' is a mnemotechnics, since its main objective seems to be the recording of memory. And yet, as I have argued earlier on in this chapter, the book is also the first (albeit constantly reworked) inscription of the system – of which the implementation is but one re-inscription. Therefore, the book *is* the system (technics). Once again, then, not only is Brooks' distinction between specifications and implementation unmade,

but neither does Stiegler's distinction between technics and mnemotechnics hold in software.

To recapitulate Brooks' argument in *The Mythical Man-Month*, the slowness that for him characterizes the process of software development, and that causes missed schedules and a reduced manageability of projects, is related to the difficulty of coordinating the different individual consciousnesses that take part in software development as a process of inscription sequenced in time. Notwithstanding Brooks' attempt to linearize the time of software development and to coordinate it through a supposedly regulated and transparent communication, the lateness of single software projects seems to be the reason for a more general slowness in the growth of software – that is, a broader slowness in technological innovation.

The latter problem is the focus of Brooks' article of 1986, 'No Silver Bullet – Essence and Accident in SE', where, rather surprisingly, he laments the *absence of technological innovation* in software. He looks back at the developments of Software Engineering between 1975 and 1985 and expresses his ultimate scepticism about the possibility of improving productivity in software. There will be no major acceleration in software growth, he prophesizes, since over a decade there has been 'no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement in productivity, in reliability, in simplicity' (181).[18] Brooks' view sounds quite opposite to Stiegler's idea that contemporary technology is problematic because the pace of technological innovation has become very fast (Stiegler 1998a). As I pointed out in Chapter One, this feeling is all too familiar to anyone who has paid attention to the rapid succession of 'generations' of personal computers or mobile phones, or to computer professionals who need to constantly 'update' and refresh their competencies just to be able to keep their jobs. The problem of obsolescence, of out-of-datedness, is key to our uncomfortable relationship with technology today, Stiegler claims. New technologies emerge and rapidly make obsolete more and more pre-existing technologies, as well as the social and cultural practices associated with them.

[18] Immediately after its publication, Brooks' article became famous precisely for destroying Software Engineering's high hopes for increasing the speed of software growth to a level comparable to that of hardware.

Therefore, there is an apparent divorce between technical and scientific knowledge on the one hand, and culture on the other. Hence the urgency of rethinking the modalities of the interaction between technology and culture. In his 1986 article Brooks also focuses on speed – but the speed of software growth is for him simply *not sufficient* (especially if compared to the fast growth of hardware). Software growth is and will continue to be slow: there are no extraordinary solutions to the problem of software slowness, no 'silver bullets'. In order to understand Brooks' position better, let me start from the metaphor of his article's title.

Two familiar themes return in the image of the silver bullet: horror and enchantment. The silver bullet hints at werewolves, which Brooks considers the most terrific of the nightmares that populate Western folklore, because 'they transform unexpectedly from the familiar into horrors' (180). What terrifies Brooks is the fact that software can turn from something familiar into a monster. He states that 'the familiar software project has something of this character'; it looks 'innocent and straightforward' but it is 'capable of becoming a monster of missed schedules, blown budgets, and flawed products' (180 f.) What I want to argue here is that Brooks' werewolf is a figure of the unexpected.[19] According to popular wisdom werewolves can be killed only by silver bullets 'that can magically lay them to rest' (180). Thus, in Brooks' theory of 1986, software has become a werewolf, a nightmare. Therefore, it must be killed, or be laid to rest. But what magic bullet can lay software to rest? Once again the unexpected takes the form of time spinning out of control (missed schedules) and of the failure to perform according to a plan (flawed products) – which in turn causes the loss of money. Consistently, the silver bullet should 'make software costs drop as rapidly as computer hardware costs do' (181).

However, the silver bullet is a magic weapon, and for Brooks it is bound to remain magic – in other words, unreal. There are no 'startling breakthroughs' in sight concerning the growth of software productivity, and, Brooks remarks, such breakthroughs are 'inconsistent with the nature of software' (181). 'Startling breakthroughs' are again a figure of the unexpected – but of a very different kind.

[19] One must be reminded here of the monstrosity of the future that Derrida upholds in *Specters of Marx*: for him, the future is monstrous, or it is no future at all (Derrida 1995).

They are welcome innovations capable of increasing the pace of software growth. Somehow, it could be said that Brooks begs to be surprised by software, but that at the same time he despairs he will never be. He actually thinks that software cannot be surprising enough – that is, that the speed of software production cannot be revolutionarily augmented. But he also states that a lot of 'encouraging innovations' are under way. 'Encouraging innovations', however, are very different from the 'alarming speed' of software growth that used to worry the participants of the first two NATO Conferences on Software Engineering (Naur and Randell 1969: 16). 'Encouraging innovations' are steady and slow efforts – similar to the stepwise progress that in the past led to finding cure for contagious diseases. The first step in medical progress, Brooks argues, was to abandon demons and humours theories in favour of germ theories – a step that 'in itself dashed all hopes of magical solutions'. At the same time, though, it turned people to a 'persistent, unremitting care' paid to 'a discipline of cleanliness'. And, he concludes, '[s]o it is with software engineering today' (181). Thus, for Brooks, finding out that there is no hope for 'magic' in Software Engineering is the first step towards a steady increase in productivity. Here magic is something unrealistic that needs to be abandoned in favour of a more enlightened perspective on software. Just as he demystified the mythical man-month in his 1975 book, Brooks now wants to demystify Software Engineering's hopes for a silver bullet.

One must be reminded at this point that albeit Software Engineering was born in the late 1960s out of the need to 'slow down' the excessive speed of software growth, just a few years later in *The Mythical Man-Month* Brooks substantially redefined the problem in terms of slowness: technology spins out of control not because it grows too fast, but because software projects are late. Thus, an interesting paradox became the foundation of Software Engineering in the 1970s: more control is necessary to prevent software from failing and projects from being delayed. Therefore, more control is needed to make software grow faster – that is, to enable and augment technological innovation. Software Engineering aims at speeding up software growth by controlling the risks implicit in that very growth. The Garmisch conference report makes clear that one has to take risks in order to make software grow – and yet, one also has to contain risks to allow for an 'ordinate' growth of software. However, in 1986, what used to be an ideal in the early days of Software

Engineering – namely, a safe step-by-step advancement – has apparently become a predicament. Software growth is so slow that the new goal is to keep going, or at least, to maintain *some* progress at all. But if control is the way to accelerate software growth (by avoiding the failure of single software projects), why has the growth of software slowed down even though the methodologies for controlling software development have become more effective?

To answer this question Brooks introduces in his article an explicitly Aristotelian distinction between the essential and the accidental – thus bringing back the separation between the ideal and the material that informed *The Mythical Man-Month*. As we have seen, such a distinction is ultimately based on the Platonic tradition that finds its most famous expression in Plato's myth of the cave *(Republic*, 514a-519a). Indeed, for Brooks, it is 'the nature of software' that prevents revolutionary leaps. Brooks defines essence as 'the difficulties inherent in the nature of the software' and accidents as 'those difficulties that today attend its production but that are not inherent' (182). For him the majority of the past gains in software productivity dealt with the accidental - they removed 'artificial barriers' such as 'severe hardware constraints, awkward programming languages, lack of machine time' (180). Nowadays, Brooks argues, software engineers do not have to deal with those kinds of problems any more, so their productivity cannot be hindered by accidental factors. What hampers productivity are inherent factors – namely, the essential, intrinsic difficulty of software as a conceptual task. Therefore, only by addressing software *as a concept* a radical improvement in productivity can be brought about, but there is still no sign of any such radical improvement.

Given Brooks' loss of hope with regard to any radical innovation in software development, it might look like software is for him incapable of generating unexpected consequences. But, after a more careful analysis, Brooks' argument offers a different interpretation: what is impossible is not the unexpected as such (since software remains fallible), but a leap of the 'right' kind - that is, a leap that results in a revolutionary increase in productivity, a welcome leap, or an expected 'unexpected'. I want to emphasize how Brooks hopes for being surprised by something he wishes for – namely, he wants to be surprised without actually being surprised. According him, what we need in order to speed up software growth is

actually *more* control – and the reason why there are no radical innovations in software is that the improvements in control achieved so far are merely of the order of the 'accidental'.

Significantly, Brooks' list of the accidental aspects of software relates the accidental to material constraints (for instance, 'machine time' is considered an accidental aspect). Even programming languages are presented as heavily influenced by the accidental characteristics of the material means of 'representation'. As I will show in a moment, Brooks actually introduces the concept of representation here – therefore abandoning the astute term 'expression' that, as I pointed out in Chapter Three, in the Garmisch conference report hints at software development as a form of exteriorization (Naur and Randell 1969: 38) - in favour of a Platonic conceptual framework which understands material objects as representing (and therefore being less real then) concepts.[20] Representation brings back the separation between content and form, concept and sign, signified and signifier. Accordingly, for Brooks the essential parts of software are 'those concerned with fashioning abstract conceptual structures of great complexity' (180). The essence of software is 'a construct of interlocking concepts' - such as data structures and algorithms – an 'abstract' entity that remains the same under many different representations (182). Brooks believes that the hard part of software development is 'the specification, design and testing of this conceptual construct, not the labor of representing it and testing its fidelity to representation' (182). Consistently with the Platonic tradition, representation is viewed here as secondary, and the separation between the ideal and the material is brought back to justify why software is difficult: software is conceptual and concepts are intrinsically difficult, while representation, the concrete, the material are trivial.[21] The inherent properties of software, in turn - its 'essence', its 'nature' – reside for Brooks in its 'complexity, conformity, changeability, and invisibility' (182). Let me now examine such properties in order

[20] As I have pointed out earlier on in this chapter, such a distinction is established by Plato in the *Republic*, 514a-519a through the well-known myth of the cave (Plato 2000).

[21] Interestingly, Brooks opposes 'syntax errors' to 'conceptual errors' (182). The syntax error - that is, a misspelled statement in a program or an error in the sequencing of statements - is viewed as banal, while conceptual errors - namely, errors in the definition of the problem/solution whole - are difficult. Here Brooks seems to forget that syntax errors can cause the general functioning of a piece of software to change significantly, sometimes with catastrophic consequences - a problem he was well aware of in *The Mythical Man-Month*, where he described software as a necessarily perfect spell.

to understand better how Brooks deploys the distinction between the essential and the accidental to justify software slowness.

As for complexity, Brooks claims that software systems are more complex than any other human construct because they contain no identical parts. If two pieces of software are identical, programmers group them into a unique piece of software and call it a sub-routine. Hardware is characterized by duplication: it contains identical parts that can be duplicated and industrially produced. Software, however, is always a one-off project – that is, each software system is developed for a specific purpose and within a specific context. Doug McIlroy's wish and enthusiasm for the industrialization of software development through the mass-production of software components (which is widely debated in the Garmisch conference report) seems to have been forgotten here.[22] It is important to notice that Brooks' claim that there is no repetition in software is in itself quite questionable, since in the mid-1980s software packages were already mass-produced. Nonetheless, it must also be kept in mind that at the time many software projects - so-called 'dedicated' or 'single-purpose' applications - were indeed still written from scratch. Even off-the-shelf software often had to be adapted to specific uses, or 'customized'. The complexity inherent in a single-purpose software system, Brooks insists, prevents its designers from having a global overview of the system. This absence of a systemic grasp in turn 'generates inconsistencies and impedes the system's conceptual integrity' (183 f.). In 1986, then, Brooks seems to have given up on his ideal that a coherent software system is the exteriorization of just one consciousness: no such perspective is possible, because software is too complex to be grasped by a single human mind. Moreover, he speaks of personnel turnover as a catastrophe. Programmers that leave the project not only subtract programming resources from it: they actually take away with them a part of the shared view of the system. The global understanding of the system, Brooks seems to imply here – the memory of the system, its history - is shared by a whole collectivity of developers. We are quite far away from his 1975

[22] As I explained in Chapter One, at the Garmisch conference on Software Engineering Doug McIlroy presented a paper entitled '"Mass-Produced" Software Components' which wanted to investigate 'the prospects for mass-production techniques in software' and recommended the creation of software components according to the same criteria that regulate the production of hardware components. For McIlroy, one important reason for the weakness of the software industry was the absence of a software components sub-industry (Naur and Randell 1969: 139). With formidable insight he supported the development of 'routine libraries' – that is, of archives of re-usable pieces of software devoted to specific functions.

view of a neat separation between the 'conception' of software (entrusted to one or two developers acting *uno animo*) and its 'realization' (split between many unthinking 'hands'). Here Brooks seems to consider the software system as the exteriorization of the memory of a collectivity.

Two other characteristics of software - conformity and changeability - are strictly correlated. In fact, Brooks claims that software always needs to conform to society, which he views as something 'cultural' and therefore 'arbitrary' (185). Brooks reasserts here the sharp distinction between technology and society that he supported in his 1975 book, and, again he describes the dis-adjustment between society and technology in terms very different from Stielger's. When investigating the historical transformation of technology, Stiegler focuses on technical systems - namely, mainly technological systems of production. In the first volume of *Technics and Time* he deploys Bertrand Gille's concept of the 'technical system' as a moment of stability in time, or a point of equilibrium in the process of technical change that characterizes history. This point of equilibrium is expressed in a particular technology. In other words, every civilisation constitutes itself around a technical system, which is in turn organized around a dominant technology. Every technical system has in itself a potential for change, and actually undergoes evolutionary trends and periods of crisis. During a crisis, the technical system evolves at great speed, causing 'dis-adjustments' with the other social systems – such as economy, politics, education, and so forth. It can only return to (relative) stability when these other systems have 'adopted' the new technical system (Stiegler 2003a: 2). However, according to Brooks, technology is the system that needs to re-adjust to society. 'In short', he states, 'the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product' (185). Furthermore, being 'soft', software is considered malleable, therefore it gets invariably changed, because the users tend to emphasise its boundaries by inventing new uses for it (185). In sum, for Brooks society poses unexpected demands to software, with which software must then comply by adapting to society. However, Brooks' view does not allow for the emergence of the unexpected - only for adjustment. Technology seems unable, in its interaction with society, to generate radical changes. What is lost here is the potential for the generation of unforeseen

consequences through the continual co-constitution and re-constitution of the technical and the social that was considered a major problem in the early days of Software Engineering.

Thus, Brooks has established that software is an inherently difficult entity because it is a complex conceptual structure that is easily manipulated by society and that must continually comply with external requests – and this inherent difficulty has not been tackled by the marginal improvements in software development achieved so far. But now he comes to what I argue is the most important as well as surprising point of his explanation of software difficulty: he declares that 'software is invisible and unvisualizable' (185). This dry proposition deserves careful analysis. In Chapter Three and throughout this chapter I have emphasized how in the discourses and practices of Software Engineering of the late 1960s and early 1970s software is understood as a material inscription made visible in different forms. Software functions *because of* and *through* being made visible - that is, its inscription, and the foundational texts of Software Engineering, actually focus on debating the best ways to make it visible. Not only the Garmisch conference report but also Brooks' 1975 book deal with this problem, although they try to keep the conceptual dimension of software separated from the material one. Here, however, Brooks qualifies software as 'unvisualizable' because it cannot be represented by 'geometric abstractions' (185). The powerful tool of technical drawing, so useful, for instance, in making inconsistencies obvious in the system specifications, he complains, cannot be used in 'software development', because software cannot be represented in two or three dimensions. It is worth noting that by 'software development' here Brooks means 'coding' (in so far as it is distinct from specifications) – and by 'software' he means 'code'. For him code – namely, computer programs - is not visualizable. Even physicists, he laments, can use 'stick-figure models of molecules' (185). 'A geometric reality is captured in a geometric abstraction', but 'the reality of software is not inherently embodied in space' (185), therefore it has 'no ready geometric representation in the way that land has maps, chips have diagrams, computer have connectivity schematics' (185). In other words, software is not comparable to a molecule, to a geographical map, or to hardware (microprocessors, computers). Brooks insists:

> As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs, superimposed one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relationships. These are usually not even planar, much less hierarchical. Indeed, one of the ways of establishing conceptual control over such structure is to enforce link cutting until one or more of the graphs become hierarchical.
>
> (Brooks 1995: 185 f.)

This passage makes clear how the combination – or rather, the superimposition - of different diagrams results in an even greater lack of clarity and in the loss of comprehensive view. Furthermore, Brooks explains that, in spite of the progress being made in simplifying the structures of software, such structures 'remain inherently unvisualizable, thus depriving the mind of some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication among minds' (186). Here Brooks pushes his Platonic framework to the limit: not only does he state that software is purely conceptual, but he even claims that it cannot be represented (at least in visual terms). Therefore, software is not easily communicable between the project participants; its invisibility undermines the very process of its development.

In sum, Brooks argues that the pace of software growth cannot be sped up because software is hard to control, hard to pin down: it is complex, always changing (in order to adjust to demands of society) and invisible - that is, non-geometric, or, in Husserlian terms, non-eidetic. Moreover, what Brooks is actually saying here is that fallibility is *essential* to software: no matter how much progress can be made at the level of the accidental, software *will always be unforeseeable 'by its nature'*. As I have mentioned earlier on in this chapter, the technical advancements made in software development are for Brooks 'accidental'. They cannot therefore substantially provide a much better control of software. Let me now examine the nature of these achievements – namely, high-level languages, time-sharing, and unified programming environments – in order to show how the distinction between

the essential and the accidental once again supports (but at the same time destabilizes) Brooks' argument for the irreparable slowness of software growth.

For Brooks, high-level languages have been the most important step toward the increase of software productivity, as well as toward its reliability and simplicity, since they *have distanced the programmer from software materiality*. He argues that high level languages 'free a program from much of its accidental complexity' (186): they leave the programmer free to deal with concepts (data structures, functions, sequences), rather than with 'bits, registers, conditions, branches, channel, disks, and such' (186).[23] It is worth noting how in this passage the locus of complexity shifts uneasily between the essential and the accidental levels of software. Even though Brooks has insisted so far that complexity is an inherent conceptual characteristic of software, if languages can free software of at least some of its complexity, then either complexity is not purely conceptual (in fact, Brooks speaks here of 'accidental complexity', although without clarifying what he means) or languages are not purely accidental. Even most importantly, he classifies 'bits, registers, conditions, branches, channel, disks, and such' as accidental (material) aspects of software. But, one could easily argue, bits, registers and so on can as well be considered abstractions. In fact, I will show in Chapter Five that every exteriorization, even the simplest distinction between zero and one, invites the questioning of the separation between the transcendental and the material. Ambivalently, Brooks considers high level languages to be further removed from materiality (which is consistent with his hierarchy between the ideal and the material) while at the same time he recognizes that '[t]o the extent that the high-level language embodies the construct wanted in the abstract program and avoids all lower ones, it eliminates a whole level of complexity that was never inherent in the program at all' (186) – therefore attributing material characteristics to language *and* to complexity. Here the boundary between what is material and what is conceptual, what is essential and what is accidental in software, becomes blurred again, and the

[23] Actually, for Brooks the main improvement in software productivity coincided with the first transition from the materiality of the machine to the symbolic level of programming. He writes: 'the biggest payoff from such languages came from the first transition, up from the accidental complexities of the machine into the more abstract statement of step-by-step solutions' (Brooks 1995: 188). He is quite sceptical as to whether newer and better high-level languages (such as Ada was at the time) would bring about any substantial increase in software productivity.

confinement of programming languages to the level of the accidental appears quite untenable.

The two other improvements in software development – namely, time-sharing and unified programming environments – are also hardly confinable to the accidental. Time-sharing is a methodology for sharing a computing resource (typically a mainframe computer) among many users. The first project to implement a time-sharing system was initiated in the late 1950s by IBM. Before time-sharing, programmers had to write a program, hand it in to the technicians who were in charge of the centralized computer, and wait for them to load and execute the program (this process was called a 'batch job') and to return the results of the execution. The turnaround time was very slow. Although Glass (2003) suggests that batch jobs gave programmers time to think, Brooks views batch jobs as constituting an 'interruption of consciousness' (187). For him batch jobs are disruptive because, by submitting their programs for execution and then waiting for results to come around, programmers inevitably forget what they were thinking when they stopped programming and called for compilation and execution (187). Therefore, they lose the general overview of the system and tend to create inconsistencies in the system itself. What I want to emphasize here is that Brooks' evaluation of time-sharing makes it quite clear that consciousness and time are constituted through technology. In a way, it could be said that time-sharing and batch jobs shape consciousness differently. Time-sharing apparently gives a more 'immediate' access to computer resources –a faster access, but a more mediated one, since programmers make use of a lot of different software packages (such as compilers, linkers and loaders) in order to execute their own programs, rather than handing it over to someone who will execute it for them and hand back the results. When time-sharing was introduced, some programmers even objected that it would lead to 'sloppiness' in programming, since people would stop thinking about their programs and start executing them all too often, in trial-and-error style. Thus, time-sharing seems to have significant consequences for the way in which software is conceptualized – that is, at the *essential* level.

Similarly, unified programming environments, seemingly preoccupied with facilitating programmers' access to integrated libraries and unified file formats,

cannot be restricted to the level of the accidental.[24] Although a detailed discussion of the characteristics of these technologies is beyond the scope of the present chapter, I want to emphasize here that unified programming environments are tightly correlated with time-sharing. Unified environments make archives of programs available to programmers. I will show later on in this chapter how the availability of programs on-line - of which the integrated programming environment was an early version - contributed to the great leap forward taken by open source programming. Brooks states that, as a result of the emergence of integrated programming environments, 'conceptual structures that in principle could always call, feed, and use one another can indeed easily do it in practice' (187). This could hardly be considered an accidental improvement. And yet, with the usual ambivalence, Brooks immediately subsumes unified environments under the aegis of instrumentality, calling them 'workbenches' (187).

To recapitulate the above argument, when he claims that software cannot be surprising, Brooks does not mean that it has no capacity for generating unexpected consequences. Software remains fallible and therefore incalculable. What Brooks rules out is the possibility of 'good' or 'welcome' unexpected consequences – in other words, of radical innovations in software and of the acceleration of technological growth. The recent improvements in software development all remain at the level of the accidental for him. They solve marginal difficulties but do not unlock the essential difficulty of software, which Brooks continues to place at the (mysterious and secluded) level of transcendence. Brooks goes as far as to examine a number of recent techniques that pretend to simplify the conceptual task of developing software in order to demonstrate that they do not bring about any substantial change. Actually, they cannot even be considered 'conceptual'. He calls them 'hopes for the silver' (188). Once again, he declares that all of them are limited to the accidental, because they are either newer high-level programming languages (such as Ada and object-oriented languages) or tools facilitating the expression of concepts (such as artificial intelligence, expert systems, 'automatic' programming and graphical programming).

[24] The first widespread examples of which were Unix and Interlisp.

What I want to argue here is that for Brooks software is predictable - that is, its incapacity for generating revolutionary improvements is predictable - precisely because it is unpredictable, which means or fallible. Software will predictably fail (in unpredictable ways), therefore projects will be late, and ultimately software growth will be (predictably) slow. What is most relevant here is that, notwithstanding his lack of hope for any revolutionary changes, Brooks reinscribes unpredictability into software. He reconfirms that software is fallible and asks for more control because he identifies the unexpected consequences of software as failures. If projects are better controlled, software will become less fallible, Brooks believes, and a step-by-step advancement will become possible (all hope for revolutionary advances having been given up). Obviously here Brooks disregards software's capacity for generating different unexpected consequences: for instance, consequences that *are not just failures.*[25]

One must be reminded here of Derrida's argument that contemporary technology is already in deconstruction. As I showed in Chapter Two, in *Echographies of Television* (Derrida and Stiegler 2002) Derrida awards this process of deconstruction quite a broad meaning. For him the acceleration of technological innovation in the contemporary world, coupled with the development of information and telecommunication technologies, constitute a 'practical deconstruction' (45) of the traditional political concepts of the public, the state, the citizen, and ultimately of the instrumental conception of technology itself. Importantly, he emphasizes how, on the one hand, in the contemporary world technological innovation is massively appropriated by multinational corporations and nation states, by means of their 'research and development' and 'defence' departments. Within this context, technological products become obsolete very quickly and technological innovations are constantly programmed to support such an economy of continual obsolescence. On the other hand, although programmed and neutralized as controlled 'development', technological innovation still gives rise to unforeseen effects. Derrida even propounds that the greater the attempt to control innovations, the more unforeseeable the future becomes. This second point is well exemplified for him by the relationships between telecommunications and the transformation of the public

[25] I will show in a moment how the open source movement is exactly one of these consequences.

space. But this point could be even better exemplified by the unexpected emergence of open source from the Software Engineering of the 1970s and 1980s (of which Brooks is one representative).

However accurate, Derrida's observations do not deal with the specificity of software's unexpected consequences – especially those consequences that lead to a change of the conception of software itself. 'Silver' is Brooks' name for 'the unexpected' – namely, the impossible revolution in software. But silver is also what is meant to kill the 'werewolf' – that is, the unexpected as failure. Thus, Brooks wishes for a bullet that would kill the unexpected – but this bullet would have to be unexpected too. Here Brooks confronts the fundamental double valence of the unexpected both as failure *and* hope. It can be said that, like Derrida's 'pharmakon', software entails both risk and opportunity, danger and cure.[26] As I remarked in Chapter Three, the risk and the opportunity implicit in technology cannot be separated. The Garmisch conference report acknowledges that risks are implicit in software, and that software fallibility is unavoidable. But it is precisely when dealing with the issue of the responsibility for the technological risk that the report establishes the impossibility of separating technology from society. Rather than as a dis-adjustment between the technical and the socio-cultural systems, the 'software crisis' at the beginning of Software Engineering can be understood better within the framework of the mutual co-constitution of the technical and the social. This is the sense of Stanley Gill's ambivalent argument that 'society' should not make 'risky' demands to technology – demands that can be met only by going beyond the current state of technology – while at the same time admitting that technology *always* entails uncalculable risks. In Chapter Three I argued that the irreconcilability of these two aspects – and therefore the necessity of calculating incalculable risks, and of attributing responsibility for them – is the point where Software Engineering 'undoes itself' precisely at the moment of its constitution. For Gill, society needs to take responsibility for incalculable risks. In fact, I want to argue that, every time we make decisions about technology, we need to take responsibility for uncalculable

[26] As Derrida remarks in 'Plato's Pharmacy', the sense of the word *pharmakon* (meaning 'remedy' as well as 'poison') remains undecidable in Plato's *Phaedrus*, and so does the boundary between memory and its supplement – a boundary that ultimately founds the devaluation of writing in Western philosophy and that constitutes 'the major decision of philosophy, the one through which it institutes itself, maintains itself, and contains its adverse deeps' (Derrida 1981: 111).

risks. There is no Habermasian way out of this irreconcilable dilemma: no 'expert' can provide society with all the necessary information to make un-risky decisions. Technology will never be calculable – and yet decisions *must* be made. But risks are also opportunities. Technology is both a werewolf and a silver bullet. Every political decision regarding technology must take into account the werewolf and the silver, the risk and the opportunity – and even the possibility that a risk might *become* an opportunity.

Let me now examine the emergence of open source in the 1990s in order to show how Brooks' risks and potential failures became unexpected opportunities for the acceleration of the pace of software growth. It is important to point out that in the late 1980s a large part of the software community agreed with Brooks.[27] To give but one example, in 1988 Robert L. Glass observed how 'No Silver Bullet' was a 'breath of fresh air' (Glass 1988: 5) because when one stops hoping for a revolution one can actually focus on achievable, evolutionary improvements. However, while the software community was breathing with relief, the quiet revolution of the open source movement was beginning to take place.

One of the most famous articles on open source and the foundational work of Software Engineering in the open source era appeared in 1997. Eric Steven Raymond's article entitled 'The Cathedral and the Bazaar' was first presented by the author at the Linux Kongress of 1997 and published as part of a book of the same name in 1999. Quite obviously Raymond's title is, twenty-two years later, a response to Brooks' book of 1975. In his article Raymond gives an account of his successful open-source project, fetchmail, which he started in 1996 and conducted as 'a deliberate test of the surprising theories about software engineering suggested by the history of Linux' (Raymond 2000: non-pag.).[28] Although Raymond's article is not deliberately presented as a contribution to the field of Software Engineering, its narrative is actually interspersed with aphorisms about effective strategies for engineering open-source software, as well as with explicit evaluations of more

[27] While in the 1975 the *Mythical Man-Month* had had a great influence but had generated little argument, in 1986 'No Silver Bullet' occasioned rebuttal papers, letters to journal editors and a lively debate (Cox 1990, Harel 1992).

[28] Linux is a Unix-like operating system started in 1991 by Linus Torvald. It is one of the most famous examples of open source software.

traditional models of Software Engineering. Let me now examine the relationship between Brooks' and Raymond's model of software development at some length.

According to Raymond, Linus Torvald developed Linux following a very different methodology from more traditional Software Engineering. To explain this point, Raymond contrasts the two models of the 'cathedral' and the 'bazaar' – where the 'cathedral' model is common to most of the commercial world, while the 'bazaar' model belongs to the Linux (and the open source) world. What Raymond calls the 'cathedral' model is in fact Software Engineering as conceived by Brooks – that is, quite a consolidated discipline with its own established corpus of technical literature. Raymond argues that the two models of the cathedral and the bazaar are based upon contrary assumptions on the nature of software development, and particularly of software debugging.

Software debugging is a late stage of software development, and is part of what in Software Engineering is generally called 'test phase' (Sommerville 1995).[29] Before being released to commercial users, a software system needs to be tested – namely, it is necessary to verify that the system meets its specifications, or (once again) that it works *as expected.* One of the activities involved in testing is debugging: when a test reveals an anomalous (or unexpected) behaviour of software, code must be inspected in order to find out the origin of the anomaly – namely, the particular piece of code that performs in that unexpected way. Code must then be corrected in order to eliminate the anomaly. The testing process takes time because all the functions of the system need to be tested. Furthermore, sometimes the correction of an error introduces further errors or inconsistencies into the system and generates more unexpected behaviour. Although in the phase of testing unexpected behaviour is generally viewed as an error, it is worth noting that decisions must still be made at this level. The testing team is responsible for deciding whether the unexpected behaviour of the system must be considered an error or just something that was not anticipated by the specifications (since, as we have seen earlier on, specifications are never complete) but that does not really contradict them. Errors need to be fixed (by

[29] Importantly, Raymond describes software development by using Software Engineering traditional terms - such as 'specifications', 'coding', 'debugging', 'testing'. These terms, whose emergence I analysed in Chapter Three, are today widely accepted and deployed in every branch of software development – not just in Software Engineering.

correcting code), but non-dangerous (and even useful) anomalies can just be allowed for and included in the specifications. Thus, the activity of deciding whether an anomaly is an error introduces changes in the conception of the system, in a sustained process of iteration.

The complexity of the above process explains why software errors are also called 'bugs'. Although the etymology of the term is uncertain, it hints at the fact that errors are often very hard to find – like the moth that Grace Hopper is said to have found trapped in a relay of the electromechanical computer Mark II in 1945, which caused many malfunctions. Locating a bug is hardly a straightforward and unequivocal process. When a malfunction occurs, it is necessary to find out what part of code causes it, and to read it in order to find out what mistake has been made in writing it. Very often no obvious mistakes (such as misspellings) can be found, because the malfunction is the result of the interaction of that piece of code with other pieces of code. Thus, more code has to be inspected, and the process tends to grow exponentially. At this point, Raymond introduces his famous aphorism that 'given enough eyeballs, all bugs are shallow' (Raymond 2000: non-pag.). I want to argue here that this principle is the foundation of the whole conception of open source Software Engineering. It certainly was the foundation of the Linux project. But in what way is this principle innovative? And in what way could Linux be considered a big leap forward in the history of software development?

In order to answer this question, it is important to notice that, as Raymond states, in 1991 nobody could have anticipated that 'a world-class operating system could coalesce as if by magic out of part-time hacking by several thousand developers scattered all over the planet, connected only by the tenuous strands of the Internet' (non-pag.). He candidly admits that he personally could not foresee it, although he had been involved in open source programming since the early 1980s, being one of the first GNU contributors.[30] Raymond considers Linux an unforeseeable change in the theories and practices of software development, which overturned much of what he thought he knew. In fact, although he was a follower of the incremental development model, he also believed that above a certain critical level of

[30] GNU is an operating system entirely based on free software. Although being Unix-like, it differs from Unix by being free software; hence its name, which is a recursive acronym for GNU's Not Unix.

complexity 'a more centralized, a priori approach was required' (non-pag.).[31] He thought that software systems as complex as operating systems needed to be built 'like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time' (non-pag.). It is worth remarking here that Raymond's metaphor of the cathedral hints at Brooks' famous theory of time management. Moreover, Raymond relates the cathedral to magic – in the sense of esoterism, secrecy and technical prowess.[32] Importantly, software as the craft of magi in splendid isolation requires that 'no beta to be released before its time' (non-pag.).

To understand this point better, one must be reminded that that a beta release (or beta version) is the first version of a software system released outside the group of developers. A beta version is released for the purpose of evaluation on the part of users (who are also called beta-testers). In the commercial world, they are usually prospective customers who receive the software for free and act as free testers. Generally beta releases are not perfect. A beta release is not a prototype – it is a finished system which has been tested and corrected to the point of performing reasonably well. Yet, developers know that it might still present malfunctions: code errors, minor imperfections, unexpected behaviour under very specific circumstances – in a word, problems that are hard to detect in the test laboratory. These problems tend to emerge only when the system is released and starts performing in a 'real' environment – that is, when it is used by non-technical users in contexts of 'real' complexity that 'stress' all the parts (or functions) of the system. Therefore, the best way to perform the fine-tuning of the system is to release it to beta users, who will interact with it from a 'friendly' perspective, knowing very well that the price they pay for the privilege of interacting with the newest version of the system is that they might encounter unexpected malfunctions. By signaling such malfunctions to the project members, once again users take an active part in the process of testing – and ultimately of software development. In commercial software, especially in the early 1990s, beta versions are released very

[31] In incremental software development a very simple version of a software system is firstly developed and more complex functions are added later on. At the beginning, the system is just capable of performing very basic tasks; then it grows increasingly complex. The main point of incremental software growth is that the system can be used from the start. In 1986 Brooks examined Harlan Mills' model for incremental software development as a 'hope for silver', but ultimately placed it at the level of the accidental.

[32] In the latest sense, Raymond's understanding of magic is closer to Gell's than to Brooks'.

cautiously – as Raymond states, 'not before their time'. The official reason is not to 'abuse the patience of the users'. In fact, the absence of beta releases is also a means of controlling the system, a form of secrecy: it prevents the users from seeing the system too early, or for seeing too much of it. By keeping the system hidden from the 'real' users, and by being the only ones who can modify it, wizards believe they protect its consistency.

Linus Torvald's style of development was the opposite of the jealous secrecy of commercial software developers. Raymond describes it in these terms: 'release early and often, delegate everything you can, be open to the point of promiscuity' (Raymond 2000: non-pag.). Finding out about Torvald's style of development was, for Raymond, a shock. Hence the metaphor: rather than a cathedral (and the 'reverent cathedral building' of traditional Software Engineering), the Linux community resembled 'a great babbling bazaar of differing agendas and approaches' (non-pag.). This passage emphasizes the disunity of the agendas of the participants in the Linux project – as opposed to the unity of intent recommended by Brooks for the good functioning of a software project (one, or at most two minds functioning *uno animo*). An example of such bazaars is, for Raymond, the Linux archive site, which 'would take submissions from anyone' (non-pag.). In fact, as we have seen before, Raymond views the fact that a coherent system can emerge from such archives as a kind of magic - or even a miracle: out of the Linux archive, he writes, 'a coherent and stable system could seemingly emerge only by a succession of miracles' (non-pag.). While the cathedral wizards are a figure of secrecy, Linux's magic is a figure of the unexpected. Here magic hints at a revolutionary leap in software growth. And yet, magic also coincides with stability. Raymond's 'miracle' is the 'coalescence' of the system, its stabilization. At the same time, Linux continued to evolve at great speed; it 'seemed to go from strength to strength at a speed barely imaginable to cathedral-builders' (non-pag.) But how are stability and speed – that is, software growth – related in the context of open source? In what way does this unexpected model of software development reconcile acceleration and the management of time, innovation and stability, software as a process and software as a product – and, ultimately, the instrumentality of software and its capacity for generating the unexpected?

In order to answer this question, let me now follow Raymond's description of the fetchmail project, which coincides with the analysis of the process of open source software development. Since 1993 Raymond had been running a free-access Internet service provider (Chester County InterLink, CCL). In 1996 he decided that he needed a new piece of software for CCL, a so-called POP3 for email management. 'So', he recounts, 'I went out on the Internet and found one' (Raymond 2000: non-pag.). As I explained in Chapter Three, this is what programmers tend to do at present: they re-use pre-existent software rather than writing it from scratch. Re-use saves time, and a piece of software that is currently being used by other programmers offers guarantees of being well-performing. Raymond theorizes re-use in one of his aphorisms: 'Good programmers know what to write. Great ones know what to rewrite (and reuse)' (non-pag.). He renames this process 'constructive laziness' and argues that it is better to start from 'a partial solution' rather than from scratch (non-pag.). In fact, Torvald did not write Linux from scratch. Rather, he started from a pre-existing software system, called Minix, and he decided to improve and enrich it with additional functions. He rewrote parts of it and kept rewriting until the original code of Minix was completely replaced by Torvald's own code. Thus, Minix was over-written - or, even better, re-inscribed - until it became a totally different system, i.e. Linux. 'In the same spirit', Raymond comments, 'I went looking for an existing POP utility that was reasonably well coded, to use as a development base' (non-pag.). He picked Carl Harris' software system popclient out of three or four pre-existing systems, because it provided 'a better development base' than the others. Raymond does not explain in detail in what way popclient's code was 'a better development base', but his text makes it clear that popclient was somehow easier to extend – in other words, that it held a *better promise for the future.*

It is worth remembering here that popclient was already an open source project – that is, a project that had been initiated by Harris but whose code was publicly available to anyone who wanted to re-use it. Raymond describes how Harris, who had lost interest in the program, handed it over to him in the typical etiquette of the open source movement. He comments that '[i]n a software culture that encourages code-sharing, this is a natural way for a project to evolve', and adds the following aphorism: 'If you have the right attitude, interesting problems will find you'

(Raymond 2000: non-pag.). Once again one must be reminded here of the early days of Software Engineering, when software was initially viewed as a problem, but then the problem itself was re-collocated (or expelled) in the world 'out there' (in society) in order to define software as a solution. Significantly, the problem that Raymond wants to solve here is itself a piece of software – the CCL system that cannot manage email properly - and the solution can be found in a different 'out there' – namely, 'out there' on the Internet – and it is a piece of software too. Moreover, Raymond understands a problem as something that actively seeks for a programmer with the right attitude to solve it, except that for him the 'problem' seems to never have been anything other than software.

Even more importantly, Raymond claims that he 'inherited' popclient from Harris. It is worth remembering here that inheriting software used to be a problem in the corporate world: earlier on in this chapter I have explained how Brooks viewed the problem of personnel turnover with horror. On the contrary, in open source inheritance is the way in which projects advance. When one is tired of a project, one can pass it on to someone else, and this keeps programmers interested and engaged. Moreover, by inheriting a software system one inherits its user base. Significantly, in open source users are considered part of the system – actually, its most important part. It can be said that users were a fundamental part of software development from the beginning. In Chapter Three I argued that the figure of the user is actually a figure of the unexpected in the Garmisch conference report. Open source explicitly conceptualizes users as co-writers of software.[33] The pre-condition for this is, of course, the availability of source code. Raymond writes:

> Another strength of the Unix tradition, one that Linux pushes to a happy extreme, is that a lot of users are hackers too. Because source code is available, they can be effective hackers. This can be tremendously useful for shortening debugging time. Given a bit of encouragement, your users

[33] Importantly, in the 1980s, while Software Engineering was preoccupied with the time management of large industrial projects, personal computers generated a mass market for software. I want to point out here that such a mass-market – which in 1986 Brooks described in terms of 'off-the-shelf software packages' – was itself an unforeseen effect of technology. Indeed, with formidable insight, Brooks commented that '[t]he development of the mass-market is ... the most profound long-run trend in software engineering' (Brooks 1995: 197).

> will diagnose problems, suggest fixes, and help improve the code far more quickly than you could unaided.
>
> (Raymond 2000: non-pag.)

Here Raymond comes to his most important conclusion: Torvald's real insight was not to underestimate the potential of the users. Therefore, Torvald did not 'invent' Linux. He rather, and more importantly, invented the Linux development model – a new model of Software Engineering. Let me now examine how this new model works and in what way it combines the speed of software growth with the stabilization of single software systems in time.

'Early and frequent releases', Raymond writes, 'are a critical part of the Linux development model' (non-pag.). As I have explained above, commercial developers do not release often, because early versions of the system almost invariably contain too many bugs, and this attitude is part of the 'cathedral' style of software development. The overriding objective in cathedral-style software development is that users see as few malfunctions as possible. On the contrary, in the bazaar-style model, users are exposed to malfunctions in order for them to contribute to finding the bugs that cause them. Imaginatively, Raymond writes: 'Linux's Internet archives were burgeoning, multiple distributions were being floated. And all of this was driven by an unheard-of frequency of core system releases' (non-pag.). Torvald treated his users as co-developers in the most effective way by organizing the time of his project according to the bazaar model: 'Release early. Release often. And listen to your customers' (non-pag.). Around 1991 (the early days of Linux) he sometimes released a new version of the system more than once a day. In sum, according to Raymond, Torvald 'didn't represent any awesome *conceptual* leap forward'; he was not 'an innovative genius' of design, but he was a genius 'of engineering and implementation' (non-pag.). This comment sounds like an implicit response to Brooks' idea that only conceptual innovation could lead to an increase in productivity of software. Torvald did not innovate at design level (which remained quite 'conservative'); rather, he innovated at the level of project organization – a level that Brooks undoubtedly considered 'accidental', and therefore incapable of generating unexpected consequences. I want to emphasize here that, while apparently deploying the rhetoric of the genius - that is, of the

exceptional creative individuality - Raymond is in fact undermining it, because ultimately the realization of an open source project is a collective task. Simply put, Torvald maximized the number of 'person-hours thrown at debugging and development, even at the possible cost of instability in the code and user-base burnout if any serious bug proved intractable' (non-pag.). This passage shows how in open source the maximization of productivity is still the aim – but now programmers are prepared to risk the instability of the system, or rather, they have accepted that instability is the fastest way forward. In a way, it can be said that open source programmers feel comfortable with the idea of working on a device that goes faster than its own time (as Stiegler would have it) and they even use such speed to manage the project itself. Torvald releases different versions of the system very rapidly, because, as Raymond explains, 'given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone', or (according to what he calls 'the Linus' Law'): 'given enough eyeballs, all bugs are shallow' (non-pag.).

Thus, we have come full circle to Raymond's initial aphorism. Indeed, Raymond states that, in a conversation with Torvald, he has developed a better formulation of this aphorism – namely, that any error will be sooner or later spotted by someone, and then someone *else* will fix it. The hard part is *finding* it. Raymond writes:

> In Linus's Law, I think, lies the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a dedicated few to develop confidence that you've winkled them all out. Thus the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect. In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena - or, at least, that they turn shallow pretty quickly when exposed to a thousand eager co-developers pounding on every single new.
>
> (Raymond 2000: non-pag.)

The main point here is that, in an open source project, there are no problems of coordination – at least, not of the order of magnitude that preoccupied Brooks. The Linus' Law can actually be rephrased as 'debugging is parallelizable' (non-pag.). Raymond explains that although debugging requires debuggers to communicate with some coordinator – that is, the developer that will actually fix the bugs, it does not require significant communication *between* debuggers. Thus, 'it doesn't fall prey to the same quadratic complexity and management costs that make adding developers problematic' (non-pag.). Moreover, the loss of efficiency due to the duplication of work by debuggers is purely theoretical. In fact, one effect of the 'release early and often' policy is 'to minimize such duplication by propagating feed-back fixes quickly' (non-pag.). Importantly, the very speed of the software growth makes up for the duplication of labour, and even for the coordination of the project. To understand this point better let me now examine how the coordination between developers is realized in open source.

For Raymond it is quite obvious that 'more users find more bugs', because they all have different ways of stressing the functions of the program (for instance, inventing new uses for it). This effect is amplified when the users are co-developers. It could be said that, in open source, making demands that exceed the boundaries of technology has stopped being a problem.[34] In fact, making unexpected demands towards software seems to be the only way for software itself to grow. And this is not just because users are now empowered with the capacity for exteriorizing the system – something that they could also do in traditional Software Engineering, although there were 'gate-keepers' who 'fended off' users' requests. More importantly, in open source the exteriorization of the software system is explicitly distributed among many individuals, who produce many overlapping re-inscriptions of the system where there are no end-to-end tasks. If the re-inscription of the system is fast enough, then it coordinates itself. What I want to argue here is that the potential of speed for managing itself is the unforeseeable consequence of the Software Engineering of the 1970s and 1980s. Indeed, a totally different point of

[34] As I showed in Chapter Three, the Garmisch conference report argues that software development reaches its point of crisis when society pushes the boundaries of state-of-the-art technology (Naur and Randell 1969: 16). However, the report was unable to determine whether such demands ultimately came from society or from technology itself. Actually, it is precisely when dealing with the issue of the responsibility for the technological risk that the report seems unable to separate technology from society.

view on speed (and on software as exteriorization) would have been necessary to foresee that, beyond a certain point, speed would become able to manage itself – or rather, that software would actually over-write itself quickly enough to be able to coordinate the very process of re-inscription.

Significantly, open source also gives an unexpected response to the problem of the individuation of consciousness identified by Brooks: in open source programmers do not need to be coordinated, because coordination 'happens' as the fast overlapping of re-inscriptions. Re-inscription involves variation. Every co-developer can potentially spot an unexpected problem in a certain software system (for instance, a certain bug) and re-inscribe the system in a different way. In what way is it then possible to stabilize a system in time? Raymond writes:

> Linus coppers his bets, too. In case there are serious bugs, Linux kernel versions are numbered in such a way that potential users can make a choice either to run the last version designated 'stable' or to ride the cutting edge and risk bugs in order to get new features. This tactic is not yet systematically imitated by most Linux hackers, but perhaps it should be; the fact that either choice is available makes both more attractive.
>
> (Raymond 2000: non-pag.)

This passage clarifies how, even in open source, software systems need to become stable at certain points in time – that is, any time one wants to stop being a developer and start being a user, one must be able to 'use' the system as a tool. The stabilization of the system coincides with its instrumentalization, or, *vice versa*, instrumentality emerges with stability in time. And yet, this stabilization is not scheduled; it is not understood as the end of a certain stage (say, specification) and the beginning of another (say, coding). In a way, there are no timetables, no deadlines. Stability is something that *happens* to the system, rather than being scheduled and worked toward. However, as Raymond notices in the above passage, a certain amount of control needs to be maintained over releases. Linux versions are numbered in order for potential users to choose which version to run. They can either run a more stable version (which nevertheless might present some anomalies that have not yet been solved) or 'ride the cutting edge' and run a newer version

(which is likely to have been further debugged and perhaps also enriched by new functionalities, but which, for this very reason, can give rise to more unexpected consequences). Raymond's passage attributes to users the capacity of evaluating the risks implicit in technology of minimizing such risks by choosing the more stabilized version of a system. Nevertheless, he has already recognized that software always entails unforeseen consequences, to the point that a system considered stable might actually lead to great surprises. I want to suggest that Raymond's distinction between risky and stable systems shows that decisions regarding technology can and must be made taking into account (rather than denying) technology's incalculability.

Ultimately, Raymond argues, open source disproves Brook's Law that 'adding more programmers to a late project makes it later' - namely, that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly. He writes:

> Brooks's Law is founded on the experience that bugs tend strongly to cluster at the interfaces between code written by different people, and that communications/coordination overhead on a project tends to rise with the number of interfaces between human beings. Thus, problems scale with the number of communications paths between developers, which scales as the square of the number of developers (more precisely, according to the formula N*(N -1)/2 where N is the number of developers).
>
> (Raymond 2000: non-pag.)

In other words, Brooks' Law rests on the assumption that the structure of communication in a software project is necessarily a complete graph - that is, that everybody must talk to everybody else. In open-source projects, this is absolutely not the case. Developers work on parallel re-inscriptions of the system and interact with each other very little. The way in which programmers communicate is ultimately the software system itself, with its continuous re-inscriptions. In a way, open source realizes the coincidence of the 'user manual' (or of documentation) with the software system.

Even more importantly, though, Raymond relates open source to a broader perspective on Software Engineering by asserting that, in order to remain interested in a project, co-developers must believe that the software system they are working on 'will evolve in something really neat and useful in the foreseeable future' (non-pag.). Once again, what programmers want is a reasonable estimate of time – only, in open source they obtain it through a fast and collective re-writing of software. Raymond writes:

> The principle behind Brooks's Law is not repealed, but given a large developer population and cheap communications its effects can be swamped by competing nonlinearities that are not otherwise visible. This resembles the relationship between Newtonian and Einsteinian physics - the older system is still valid at low energies, but if you push mass and velocity high enough you get surprises like nuclear explosions or Linux.
>
> (Raymond 2000: non-pag.)

One could hardly use a more poignant image. Here open source programming is explicitly defined as the unexpected: it is the surprising explosion of something new, something that becomes visible only by virtue of its very speed.

To conclude, in this chapter I have attempted to demonstrate how open source programming emerges as the unforeseen consequence of the more traditional Software Engineering. The sequencing of time that Software Engineering proposed in the 1970s and 1980s, and justified through a (ultimately Platonic) distinction between the ideal and the material, eventually gave rise to an unexpected re-organization of technology according to the open source style of programming. The framework of instrumentality that, as I showed in Chapter Three, emerged from the 'software crisis' of the late 1960s - that is, from the need to control the excessive speed of software growth - was enforced in software production through the 1970s and 1980s, and it enabled the development of software during these decades *as well as* the unexpected emergence of open source. Open source still has Software Engineering as its model, of which Raymond's theory is the most famous example. In this model the framework of instrumentality is once again re-enacted: the aim of open source is still to obtain 'usable' software, but 'usability' - that is, a stable

version of a software system - is not something that can be scheduled in time. Rather, it is something that *happens* to the system while it is constantly re-inscribed. Software Engineering is characterized by the continuous opening up and reaffirmation of the instrumentality of software (and ultimately of technology). In Software Engineering, the instrumentality of software is (as Derrida would have it) 'in deconstruction': it is the unstable result of the process of technological exteriorization. And yet, since the undoing and re-doing of instrumentality can go unnoticed, a 'deconstructive reading' - namely, a problematization - of software must be actively performed. This active problematization of software ultimately makes 'originary technicity' most visible. In other words, it clarifies the significance of software (and of singular instances of software) for the relationship between technology and the human.

5 Writing the Printed Circuit

For a Genealogy of Code

DO 10 I = 1.10

(FORTRAN line of code said to have caused the crash of the Mariner I space probe in 1962)

In Chapter Four I argued that in the late 1970s and early 1980s the discipline of Software Engineering reasserted its capacity for continually opening up and preserving the instrumental conception of software (a capacity that can already be found in the early days of Software Engineering at the end of the 1960s). I also suggested that the emergence of open source programming in the 1990s, with its own brand of Software Engineering practices and theories, can be understood as one of the unforeseen consequences generated by the conception of software developed in the 1970s and 1980s. In this chapter I want to take this argument further by showing how software escapes instrumentality not only by generating unforeseen consequences, but also *by repeatedly challenging the very process of linearization through which it is constituted.* As discussed earlier, linearization is deeply correlated with instrumentality. In fact, it is precisely by becoming linearized – that is, based on the phonetic alphabet – that writing becomes instrumental to spoken

language within the Western philosophical tradition.[1] By becoming a medium for the phonetic recording of speech, writing at the same time becomes a technology. To be more precise, it is situated at the level of the tool. It can therefore be suggested perhaps that linearization constitutes instrumentality – or even that instrumentality is constituted *through* and *as* linearization. And yet, as Jacques Derrida points out in *Of Grammatology* (1976), linearization is nothing but the constitution of the 'line' as a norm – in other words, the line is *only* a model, however privileged. In what follows I intend to demonstrate that such an unstable process of linearization lies at the very basis of the relationship between 'software', 'writing' and 'code' the way they are conceived in Software Engineering. To do this, I will stray a little from the technical literature of Software Engineering in its strict sense in order to investigate the development of programming languages in the late 1960s.

As explained in Chapters Three and Four, implementation – or the translation of software specifications, however detailed and formalized, into code (i.e., computer programs, or texts written in a programming language) is an important stage in the process of software development. By looking at the development of the concept of 'programming language' and of its relationship with the concepts of 'software' and 'code', I want to show that in the late 1960s programming languages were constituted as linear constructs in order to make software instrumental – or that software has been constituted in the form of the linear sequencing of symbols in order to be thought of as a tool. In the late 1960s, in the early days of Software Engineering, quite a large number of high-level programming languages were already in use, and many more were being developed and gradually becoming

[1] The French paleonthologist André Leroi-Gourhan defines the emergence of alphabetic writing as the 'linearization' of a pre-existent form of graphism tightly correlated to figurative art (Leroi-Gourhan 1998: 190-216). He links the emergence of alphabetic writing to the technoeconomic development of the Mediterranean and European group of civilizations. At a certain point in time during this process writing became subordinated to spoken language. As a tool, its efficiency became proportional to what Leroi-Gourhan views as a 'constriction' of its figurative force, pursued precisely through an increasing linearization of symbols. In the context of his own reflection on writing in *Of Grammatology* Derrida draws on Leroi-Gourhan's view of phonetic writing as 'rooted in a past of nonlinear writing', and on the concept of the 'linearization' of writing as the victory of 'the irreversible temporality of sound' (Derrida 1976: 85). Expanding on Leroi-Gourhan, Derrida relates the emergence of phonetic writing to a linear understanding of time and history. Linearization is nothing but the constitution of the 'line' as a norm. Yet, the line is *only* a model, however privileged. Questioning the idea of language as linear therefore implies questioning the role of the line as a model, and thus the concept of time as modeled on the line. It also implies questioning the very foundations of Western thought. This is why a rethinking of technology (which is related to language and writing) entails a rethinking of Western philosophy.

available - such as FORTRAN, COBOL, PL/1 and Algol. The relative value of some of these languages was actually a topic of discussion at the two NATO Conferences on Software Engineering (Naur and Randell 1969, Buxton and Randell 1970, Randell 1979, Sebesta 2008). Nevertheless, and even more importantly, the foundation of *the theory of programming languages* - namely, the general theorization of what a programming language ought to be - was being laid out systematically only at the very end of the 1960s.[2] In what follows I will look at one of the foundational texts of the theory of programming languages – i.e., *Formal Languages and Their Relation to Automata,* published by John E. Hopcroft and Jeffrey D. Ullman in 1969 (the year of the second NATO Conference on Software Engineering) - in order to investigate to what extent and in what way the process of 'linearization' can be shown to be at work in the constitution of the concepts of 'programming languages', 'software' and 'code'. I want to focus on how professionals and academics struggled with the idea of linearization in order to make software manageable. The instrumentalization of technology and the linearization of programming languages go hand in hand in the theory of programming languages, and therefore the problematization (or, as Derrida would have it, the deconstruction) of one leads to the problematization of the other.

By doing this, I also want to take further my analysis of the relationship between software and time. The latter is understood here not as the time of software development (the way I have defined it in the previous chapters), but as the linearization of time implicit in the theory of programming languages. In Chapter One I investigated the constitutive relationship between technology and temporality. Specifically, following Derrida's and Stiegler's work, I examined how the 'mark' constitutes temporality. For Stiegler, '[t]hat which makes consciousness be self-consciousness (i.e. consciousness that is conscious of contradiction with itself) is the fact that consciousness is capable of externalising itself' (Stiegler 2003:163) – for instance, the fact that one night one can write the sentence 'it is dark' and then reread it twelve hours later when it is light, thus entering into a 'dialectic' with oneself. This 'contradiction between times', namely the time of the consciousness

[2] Although the theory of programming languages needs to be seen in its singularity, its fundamental concepts have remained the same to date, and are still taught in foundational courses on programming languages at undergraduate and postgraduate level.

when one wrote this sentence and the time of the consciousness when one reads it, puts in crisis consciousness itself. This crisis, in turn, raises one's self-awareness. What both Stiegler and André Leroi-Gourhan call 'exteriorization' through writing is precisely what constitutes consciousness. The act of inscription - of exteriorization - ultimately enables the emergence of interiority (which does not precede exteriority, and vice versa). The very process of exteriorization constitutes the foundation of temporality, of language and of technology. We may remember that in Chapter Three I described the process of software production as the exteriorization of consciousness through writing - or through the material inscription of software. Yet the process of inscription is not just what produces consciousness (as awareness of time and therefore self-awareness): it is also prior to every distinction between the transcendental and the empirical, the ideal and the material. In fact, the ideal can be constituted only through the material - through empirical repetition in time and space. We should bear in mind that even the difference between '0' and '1' can be constituted *only through a process of empirical repetition in time and space*. The distinction between 0 and 1 is always already material, since it is enabled by what Derrida calls 'the instituted trace', or the possibility of making conceptual distinctions through time, and marking these distinctions somehow – be it in our memory or through some form of mnemotechnics. By considering the distinction between 0 and 1 as 'transcendence enabled by materiality' (as *all* transcendence is), one can break free from Hayles' dilemma regarding the ontological status of code, since the material transcendence that enables the distinction between '0' and '1' is the same one that enables metaphysical thought, including ontology.[3] Moreover, since consciousness is constituted through repetition – because writing registers the past, thus enabling a different relation to time and making transcendence possible - we can say that conscience and time are constituted by the very possibility of the distinction between '0' and '1'. It can therefore perhaps be suggested that *we have always been*

[3] As I have mentioned in Chapter Two, Hayles (2005) raises the question of the ontological status of code, and suggests that code reduces its ontological presuppositions to a minimum – that is, to the very distinction between '0' and '1'. From this point of view, the distinction between '0' and '1' is not the minimization of the ontological presuppositions of code (and therefore of code's relationship with metaphysics), but the very foundation of thought, including metaphysical thought and the very possibility of making ontological distinction.

digital.[4] This insight allows us to see technology as ultimately constitutive of thought (including philosophical thought). This is why technology can never be definitively expelled from thought. In order to think technology in a politically meaningful way it is necessary to keep in mind that technology is constitutive of consciousness and of temporality – and thus also of our being human. I will return to this point toward the end of this chapter. For now, let me take a closer look at the state of the art of programming languages at the end of the 1960s.

As mentioned above, the late 1960s were not only the time of the first two conferences on Software Engineering; they were also the time of substantial changes in the development of programming languages. By 1968, there existed a vast number of high-level programming languages, albeit only about fifteen were in widespread use, the most common being FORTRAN and COBOL. To put this in context, one must bear in mind that the first digital computers appeared in the 1940s and were used for scientific applications. As Robert W. Sebesta states in his brief history of programming languages, '[t]he early high-level programming languages invented for scientific applications were designed to provide for those needs' (Sebesta 2008: 5), which throughout the 1950s and 1960s basically coincided with the need for efficient floating-point arithmetic computation.[5] However, the use of high-level languages was not yet consolidated in all areas of computing, as the debate at the NATO Conferences on Software Engineering showed (Naur and Randell 1969, Buxton and Randell 1970). In fact, the other main area of interest at the time of the NATO conferences was related to the commercialization of computers, and thus to 'systems programming'. One of the most significant issues of the time involved having to decide whether operating systems and all the programming support tools of a computer system, collectively known as *systems software*, which were still being provided by manufacturers for free, should be priced and commercialized separately. This debate was also concerned with what

[4] 'We Have Always Been Digital' is the title of Joanna Zylinska's ongoing photographic project that explores digitality as the intrinsic condition of photography in the present *and* in the past (http://photos.joannazylinska.net/).

[5] The question of the accuracy of calculation performed on a computer is very much a physical problem. Scientific applications typically have to deal with very large or very small numbers, whose length easily exceeds the fixed memory space for numerical data. Therefore a microprocessor and a programming language devoted to scientific applications must be able to store numbers in a way that takes into account all significant digits – that is, a way in which the decimals are not fixed but 'floating'. The first language ever developed for scientific applications was FORTRAN.

languages were best suited for systems programming. Being used almost continuously, systems software needed to be efficient; therefore, it had to be written in programming languages that could be executed quickly. Furthermore, such languages had to provide 'low-level features' – or instructions that allowed programmers to access and reprogram very specific areas of the computer memory and circuitry. An important controversy at the NATO conferences was whether it was possible to use high-level languages for systems programming (Randell 1979: 3).[6] Many participants in the NATO conferences deemed it impossible; they actually claimed that they preferred languages with 'low-level features' because they wanted to keep 'close to the machine'. As I showed in Chapter Three – and as will become clearer throughout the course of this chapter - the distinction between programming languages and 'the machine', or between language and materiality, was already quite unstable; it was, so to speak, 'in the process of making and unmaking'. The general anxiety about losing touch with materiality could be interpreted as the fear of losing the ability to influence the functioning of computer memory and circuits. Again, this point will become clearer later on in this chapter, when I investigate how programming languages 'translate' into circuitry. However, in the 1960s and 1970s the 'software scene' (to use Randell's term) saw the development of special machine-oriented high-level languages. These languages were meant to be used for writing systems software for different machines produced by different computer manufacturers (for example, PL/S was developed to write systems software for IBM mainframe computers). Finally, it is worth remembering that the issue of programming languages concerned not just systems software but also real-time applications – that is, applications where time was critical, such as military

[6] Many participants in the NATO Conferences were also involved in the design of new programming languages. One of the main technical developments at the time was in fact the design of ALGOL (with the watershed meeting in Munich held not long after the Garmisch conference) and its competition with FORTRAN (in which IBM had a vested interest that ultimately led to ALGOL's failure on the market) and Pl/1 (a 'universalistic' language whose development was the result of a certain entrenchment of interests between business and scientific communities) (Sebesta 2008: 56 ff.).

applications, which were most likely at the core of NATO's interest in programming languages.[7]

In sum, high-level languages were a reality at the end of the 1960s. Hopcroft and Ullman published their work on languages and automata in 1969. Although the theory of formal languages was already a dynamic subfield of computer science, this book took it one (substantial) step further. It constituted a systematization of the authors' research in the field, previously disseminated in journals of mathematics and computer science and in notes for courses on the theory of formal languages that the authors had taught at Princeton, Columbia and Cornell University (Hopcroft and Ullman 1969: vi).[8] The aim of the book was to present 'the theory of formal languages as a coherent theory' and to make explicit its relationship to automata

[7] Other areas of computing were left untouched by this debate: for instance, business applications worked reasonably well at the time and were dominated by a specific high-level language, COBOL, whose first implemented version dated from 1960 and which had just achieved the status of a *de facto* standard in the United States (Randell 1979). It must be noticed, however, that the first formal meeting for the definition of a 'universal' language from business application (which ultimately led to the design of COBOL) was also sponsored by the American Department of Defense, and it was held at the Pentagon on May 28 and 29, 1959. The meeting established that the prospective programming language should use English as much as possible (in order to allow managers to read programs) and be easy to use (to broaden the range of those who could program computers). The initial specifications for COBOL were published by the Department of Defense in 1960, and revised versions were published in 1961 and 1962 (Department of Defense 1960, 1961, 1962). The language was standardised by the American National Standards Institute (ANSI) in 1968, and subsequently revised in 1974, 1985 and 2002. The language continues to evolve today and it may still be the most widely used language: for instance, in the late 1990 a study estimated that there were approximately 800 million lines of COBOL in use in the 22 square miles of Manhattan (Sebesta 2008: 62).

[8] John Edward Hopcroft taught at Cornell University in 1969. His books on formal languages, coauthored with Jeffrey D. Ullman (and later with Alfred Aho) are regarded as classics of the field. In 1969 Ullman taught at Princeton University, after having worked for the Bell Labs for several years. In 1986 Ullman co-authored with Alfred V. Aho and Ravi Sethi the famous textbook entitled *Compilers: Principles, Techniques, and Tools*, which as of today is still widely regarded as the definitive text on compilers design. Interestingly, Aho, Sethis and Ullman seemed to be obsessed with Christian figurations as much as Fred Brook was (as noted in Chapter Four; see also Brooks 1987, 1995) – quite appropriately, given their understanding of computing remains within the terms of the Platonic/Aristotelian tradition. In fact, *Compilers: Principles, Techniques, and Tools* is known as the 'dragon book' because the first edition had a picture of a dragon and a knight on the cover, which was meant – the authors explained – as a metaphor for conquering complexity. The dragon was kept in the second edition, which involved Monica S. Lam as a fourth co-author. The first edition became known as 'the red dragon book' (also to distinguish it from Aho and Ullman's *Principles of Computer Design*, published in 1977, which in turn had a green dragon on its cover and was therefore nicknamed 'the green dragon book'). The second edition (published in 2006) of *Compilers: Principles, Techniques, and Tools* is known as the 'purple dragon book' (Aho et al. 2007).

(v).[9] It represented the most synthetic account of the theory of formal languages in 1969 – a theory that Software Engineering relied on (and took for granted) when conceptualizing the stage of 'implementation' in the process of software development. As I have pointed out earlier, although the foundations of the theory of formal languages have remained unchanged to a large extent over the decades (and although this theory still constitutes the basis of the programming languages in use today), Hopcroft and Ullman's book needs to be studied in its singularity.[10] Nevertheless, in the rest of the chapter, whenever Hopcroft and Ullman's account is so synthetic that it risks sounding unclear, I will refer to some later texts, and especially to Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman (2007), to clarify concepts.

The theory of formal languages sprang to life in the mid-1950s, when Noam Chomsky developed a mathematical model of what he called a 'grammar' in connection with his study of natural languages (Chomsky 1956). In the preface to their key text, Hopcroft and Ullman comment:

> Shortly afterwards, the concept of a grammar was found to be of great importance to the programmer when the syntax of the programming language ALGOL was defined by a context-free grammar. This development led naturally to syntax-directed compiling and the concept of a compiler. Since then a considerable flurry of activity has taken place, the results of which have related formal languages and automata theory to such an extent that it is impossible to treat the areas separately. By now, no serious study of computer science would be complete

[9] An automaton is basically a formalism used to describe abstract systems and their transformations (Aho et al. 2007: 147). I will not enter into the detail of the theory of automata, since for the purposes of this chapter I focus on the concepts of 'language' and 'grammar' in the theory of formal languages. Nevertheless, in the second section of the chapter I will relate formal languages to automata and I will show how the theory of automata is especially important for compilers – that is, for programs that translate other programs to make their execution possible.

[10] The languages I have addressed earlier, such as FORTRAN, COBOL, PL/I and ALGOL (but also more recent languages such as C and Pascal) are all based on the theory of formal languages (and more specifically on a particular formalism known as the Backus-Naur Form). The same is true for languages related to Artificial Intelligence, which were also a reality in the late 1960s, having started out with LISP in 1959 (McCarthy et al. 1965), which was followed by Prolog in the early 1970s (Clocksin and Mellish 2003). The so-called 'Web Software', which has emerged much more recently, still comprises languages based on that same formal theory (such as Java, which draws heavily on C), as well as 'markup languages', such as XHTML, which strictly speaking is *not* a programming language (Sebesta 2007).

without a knowledge of the techniques and results from language and automata theory.

(Hopcroft and Ullman 1969: v)

I will clarify the technical terms used in this passage later on in this chapter. For now suffice it to say that, although Chomsky aimed at providing a formal description of *natural* languages, the most powerful and productive influence of his theory is to be found in the field of *programming* languages. In fact, the circularity of Hopcrof and Ullman's argument here seems to constitute a 'point of opacity' in the theory of formal languages. Rather than to actually explain natural languages, formal grammars (also known as 'Chomsky's grammars') are best deployed in modelling programming languages – and this is because programming languages are designed *accordingly* to formal grammars.[11]

In order to clarify this point, let me follow Hopcroft and Ullman's argument for a little while. The book begins by defining the concept of a 'finite description' of a language. This definition serves as the foundational concept of the theory of formal languages. The Chomskyan concept of 'grammar' is established as the fundamental descriptive device for formal languages.[12] It must be emphasised here that Hopcroft and Ullman consider the pre-Chomskyan definitions of language (for instance, Webster's definition of language as 'the body of words and methods of combining words used and understood by a considerable community') as not sufficiently precise for computing. The goal of Hopcroft and Ullman's book is thus to 'define a formal language abstractly as a mathematical system' (1). In other words, in order to be useful in computing, a language needs to be abstract, or formalized according to a mathematical model.

One must be reminded at this point that the association between formalization and instrumentality goes back at least to Husserl's reflections on geometry. Husserl

[11] As I explained in the Introduction, from the perspective of deconstruction a point of opacity is a point where a given conceptual system 'undoes itself' – that is, a concept that has to remain unthought within that conceptual system in order for the system to exist (Derrida 1980).

[12] In the initial section of their book Hopcroft and Ullman detail what a grammar is, introduce its three major subclasses (regular, context-free and context-sensitive grammars) and single out 'context-free grammars' as the foundation of the theory of formal languages.

associates algebra, or the formalization of geometry, with instrumentality (what Stiegler would call 'the technicization of science'), and, after Plato, with the loss of memory. For Husserl, by becoming a form of calculation, geometry renounces its capacity for visualizing geometrical shapes, or 'spatio-temporal idealities' (Husserl 1970: 41). Algebra, as a technique of calculation, is nothing but a formalism that allows us to manipulate numerical configurations and to forget their visual meaning. From this point of view algebra is devalued; everything formal is devalued as 'eidetically blind' (in Stiegler's terms) – in other words, as instrumental, or 'just technical'. From this point of view, Hopcroft and Ullman's attempt to formalize language - or, better, to construct a formal theory of languages – is also an attempt to make language instrumental. Formal languages (and therefore also programming languages), it can be said, are constituted *as* instrumental. In the remaining part of this chapter, I intend to demonstrate, however, how this formalization is simultaneously established and undone in the theory of formal languages. In order to do this, the very idea of 'abstraction' must be looked at more closely.

In Chapter Two I raised some questions for the notion of abstraction by following Jacques Derrida's re-reading of Ferdinand de Saussure's structural linguistics. Since his earliest works (in particular *Of Grammatology*), Derrida has defined writing as a material practice. For him 'writing' takes precedence over orality not because writing historically existed *before* language, but because we must have a sense of the permanence of a linguistic mark (a sense of its inscription, of its being embodied in a material surface) in order to recognise it.[13] This is what Derrida means when he says that 'the transcendental' is always impure, always already contaminated by 'the empirical'. Thus, textuality and materiality are not opposed: materiality is the condition of signification and every code is material – or, to be more precise, the possibility of inscription is the very condition of code's functioning. But if every code is material, and if the material structure of the mark is at work everywhere, in what way can languages – and especially programming languages - be described in 'abstract' terms? What does it mean to construct, in

[13] In other words, although we recognize the written form of a grapheme (let's say 't') only by abstracting it from all the possible empirical forms a 't' can take in writing, we need such an empirical inscription to make this recognition possible.

Hopcroft and Ullman's terms, an abstract theory of language? In order to answer this last question, we must examine Hopcroft and Ullman's definition of language.

Hopcroft and Ullman provide the following definition of an alphabet: '[a]n *alphabet* or *vocabulary* is any finite set of symbols', and then add: '[a] set is countably infinite if it is in one-to-one correspondence with the integers (i.e., if it makes sense to talk about the *i*th element of the set)' (1). I want to suggest here that Hopcroft and Ullman's characteristic of 'countability' coincides with what Katherine Hayles names 'discreteness'. She considers it one of the four specific characteristics of code, which for her set code apart from 'writing' and 'speech' (Hayles 2005: 56). According to Hayles, discreteness is actually a synonym for 'digitization' – and, conversely, digitization is the operation 'of making something discrete rather than continuous, that is, digital rather than analog' (56). She also relates digitization to materiality by explaining that '[t]he act of making discrete extends through multiple levels of scale, from the physical process of forming bit patterns up through a complex hierarchy in which programs are written to compile other programs' (56). For her digitization 'happens' in the inner circuitry of a computer when the bit stream is formed from changing voltages channelled through logic gates. Digitization can hardly be found in speech and writing and therefore it is hardly mentioned by Saussure or Derrida, whom she considers the fundamental thinkers of (respectively) language and writing. In fact, although Hayles acknowledges the importance of the 'blank' in Derrida's grammatological thought, and agrees that spaces play an important role in the digitization of writing, she does not relate the blank to software. She simply points out that Derrida's spacing was what made writing not just a simple transcription of speech but rather something that exceeded speech. In the same way, she states, code exceeds both speech and writing, and cannot be encapsulated in them (57). And yet, as I will show later on in this chapter, Derrida's understanding of the 'blank' has much deeper consequences for the study of software than Hayles acknowledges. But before I develop this point, let me follow Hopcroft and Ullman's analysis of the alphabet a little further. For them, alphabets are *finite* sets of symbols which

include digits, the Latin and Greek letters both upper and lower case (possibly with combinations of subscripts, superscripts, underscores, etc.), and special symbols such as #, ¢, and so on. Any countable number of additional symbols that the reader finds convenient may be added. Some examples of alphabets are the Latin alphabet, {A, B, C, ..., Z}, the Greek alphabet, {α, β, γ, ..., ω}, and the binary alphabet{0, 1}.

(Hopcroft and Ullman 1969: 1)

The first important aspect to be noticed here is that this is a *written* alphabet, since it is case-sensitive - that is, it distinguishes between upper and lower case - and it includes a number of special symbols with no phonetic equivalent. It is also clearly a Western-centric one, because it includes Greek and Latin characters.[14] Even more importantly, Hopcroft and Ullman give examples of what they call 'natural' alphabets – which I want to rename here as alphabets associated with 'natural' language. We have to remember, however that, as I argued in Chapter One, no 'natural' alphabet actually exists: any alphabet is always already technological.[15] Furthermore, the concept of the alphabet – including alphabets associated with natural languages – entails a form of linearization. Even more importantly, it is worth noting that, along with the 'natural' alphabets, Hopcroft and Ullman introduce the notion of the 'binary alphabet'. In this extremely relevant passage, binary notation (which is the foundation of digitization) is presented *as just another alphabet*, no more and no less (un)natural than all the other possible alphabets, and containing just two symbols. Therefore, here *digitization is presented as an alphabet – that is, as a process of linearization.*

Having defined the alphabet, Hopcroft and Ullman proceed to define the 'word'. They write: 'A s*entence* over an alphabet is any string of finite length composed of symbols from the alphabet. Synonyms for sentence are *string* and *word'* (1). 'String' is one of the most commonly used terms in the theory of programming

[14] Other important examples of alphabets are the ASCII alphabet (which is used in many computers) and Unicode, which includes roughly 100,000 characters from all over the world (Aho et al. 2007: 117 f.). As it will become clearer in a moment, Unicode constitutes an especially poignant attempt at what I would call a 'universal' linearization.

[15] Again, the association between the alphabet and technology is developed by Leroi-Gourhan (1993) and subsequently re-read by Derrida (1976) in the context of his own reflection on writing.

languages.[16] It is worth noting here that this definition is circular again, since a sentence is defined as a string, but then 'string' is given as a synonym for 'sentence'. However, although the 'string' itself is not explained further, what has been said about the alphabet makes it obvious that a string is a finite, ordered sequence of discrete symbols drawn from the alphabet (see also Aho et al. 2007: 118).[17] I want to emphasize here that the concept of the string makes it even clearer how the theory of formal languages (and specifically the kind of *formalization* of languages that it propounds) works as a process of *linearization*. Moreover, the definition of the string is an important part of the process of linearization through which software is constituted.[18]

To understand this point better, let me now give one example of Hopcroft and Ullman's string by focusing on what they name the 'binary alphabet' and define as {0, 1}. Starting from this alphabet, for instance, one can construct the following string: 00110110001101.[19] We have to bear in mind that the binary alphabet is a very small alphabet with which an infinite number of strings can be produced. As I suggested at the beginning of this chapter, one very important point that can be drawn out of this definition of the binary alphabet is that, rather than minimizing *ontological* requirements - as Hayles (2005) claims - computation actually minimizes *alphabetic* requirements. Significantly, for Hopcraft and Ullman natural and programming languages are not as different as they are for Hayles. Even more importantly, Hopcroft and Ullman's definition of computation as an alphabet seem

[16] However, the word 'word' is very significant, because it has two fundamental meanings in computer science. The first one is that proposed by Hopcroft and Ullman. The second – as provided, for instance, by the *Webster's Dictionary of Computer Terms* – is 'a unit of information composed of characters, bits, or bytes that is treated as an entity and that can be stored in one location'. This second concept is considered hardware-related; so much so, in fact, that the 'word size' is taken as a unit of measure for computer hardware, and it expresses the number of bits a computer can work with at one time. Hopcroft and Ullman ignore this second meaning – a meaning that Hayles would perhaps qualify as 'more material', but that I would take as a striking example of the materiality of the mark. I will return to this second sense of the word 'word' later on in this chapter.

[17] Obviously Hopcroft and Ullman's 'symbol' must not be understood as pertaining to the realm of the 'symbolic' in the traditional philosophical sense that opposes the 'symbolic' to the 'real' or to the 'material'. For Hopcroft and Ullman the concept of 'symbol' corresponds to a character, a digit or another graphic sign - in a word, to a mark – and is mainly characterised by its discreteness.

[18] I will show later on in this chapter that a computer program, written in a programming language, can be viewed as a string. This idea of the program as a sequence echoes Leroi-Gourhan's concept of the *chaine operatoire*. The *chaine operatoire*, (or 'operating sequence') is for Leroi-Gourhan the kind of sequential organization that underlies both language and technology (Leroi-Gourhan 1993). For him, both language and technology are characterized by the presence of an organizing 'syntax' which sequentially organizes gestures, tools and language.

[19] If the alphabet is, say, the set of characters of the English alphabet, one example of a string would be 'hello'.

to break free from the dilemma of the ontological status of code that Hayles herself sets up by claiming that code reduces ontological requirements to a minimum. What I want to emphasize is that, rather than taking sides on the ontological status of code - that is, rather than asking what code *is* - I am examining here Hopcroft and Ullmann's singular definition of computation *as* an alphabet. In other words, I am examining a singular answer to the following question: 'How does software work?'. Hopcroft and Ullman place all languages in a sort of continuum, where it is possible to define an appropriate alphabet for each language and to construct the sentences (or strings of symbols) that belong to that language. For them the difference between 'natural' and programming languages seems to reside in the different kinds of 'grammars' that are necessary to 'describe' these languages. The continuum between natural and programming languages is based on linearization as well as on the technological nature of the alphabet and – more importantly – of consciousness. As I have explained at the beginning of this chapter, the possibility of distinguishing between '0' and '1' coincides with the possibility of making *all* conceptual distinctions (including those regarding ontology). To deploy Hayles' terms, it could be said that the point is not the (supposedly minimal) relationship between ontology and code, but the (constitutive) relationship between technology and thought.

In sum, we have ascertained so far that the 'string' is a sequence composed by symbols belonging to a given alphabet. Besides, Hopcroft and Ullman define the empty string in the following way: '[t]he empty sentence, ε, is the sentence consisting of no symbols' (1). The concept of the empty string is motivated by the needs of the formal system to work.[20] It is also important to notice that any string *s* has a finite length, generally indicated with $|s|$, which is the number of occurrences of symbols in *s*. For instance, *banana* is a string of length six. The empty string ε is the string of length zero (Aho et al. 2007: 118). At this point, the authors can give their first formal definition. They write: 'If V is an alphabet, then V* denotes the set of all sentences composed of symbols of V, including the empty sentence. We use V^+ to denote the set V* - {ϵ}. Thus, if V = {0, 1}, then V* = {ϵ, 0, 1, 00, 01, 10, 11, 000, ...} and V^+ = {0, 1, 00, ...}' (Hopcroft and Ullman 1969:1).

[20] To build a working formal system, a 'neutral' element is needed. It becomes clear at this point that Hopcroft and Ullman's theory of formal languages is based on set theory.

It is not so important now to focus on the difference between V* and V^{+}. What we need to keep in mind is that a string 'over an alphabet' (Hopcroft and Ullman 1969: 1) can be any finite sequence or string of letters belonging to that alphabet, and that a 'language' is any discrete set of strings 'over' some fixed alphabet. First of all, it is worth giving some consideration to the word 'over', which Hopcroft and Ullman use regularly. The word 'over' defines the alphabet as a *space* - in fact, a linearized space, or rather, the space *of* linearization. A language can be defined only 'over' an alphabet - in other words, somehow 'within' it and 'in terms of' it. However, Hopcroft and Ullman's is a very broad definition of 'language'. Totally abstract languages such as an empty language (ø, the empty set) or {ε} (the set containing only the empty string) count as languages under this definition. So does the set of all grammatically correct English sentences or the set of all syntactically correct programs in C, or in any other programming language – although, as we shall see in a moment, it is not so easy to specify what these languages are. Importantly, this definition of language does not require any meaning to be ascribed to the strings in the language. To specify the meaning of the strings, further methods are necessary (Aho et al. 2007: 118).[21] It must also be noticed that the notation V* uses the so–called 'Kleene star' – that is, an operator that, starting from an alphabet, builds all the possible strings over that alphabet.[22] Therefore, V* is an infinite set of strings, each of which is of finite length. Moreover, by concatenation, or by linking together two or more strings (one immediately after the other), it is possible to form a new string, whose length is the sum of the lengths of the original strings. This operation is defined by Aho et al. (2007) as 'appending' a string *x* to a string *y*; for instance, if *x=dog* and y=*house*, *xy=doghouse* (Aho et al. 2007: 119). The result of concatenating a string with the empty string is the original string – therefore, 'the empty string is the identity under concatenation' (118). These two operations make the process of linearization even more visible. Yet, I want to highlight here that the Kleene star is not the representation of the process of linearization, but just one possible operation that can be done within an already linearized system of reference. The Kleene star is meant to work in presence of discreteness - in other words, of the linearization that has always already occurred in the alphabet - and it

[21] I will come back to the problem of the 'semantic' correctness of a program when I discuss compilers later on in this chapter.

[22] Briefly put, an operator, such as '+' or '/', is a symbol that describes a (usually logical or mathematical) function, such as adding or dividing.

produces repetitions of discrete elements. In fact, it is also called the 'any-number-of' operator (Aho et al. 2007: 187). Tellingly, Hopcroft and Ullman eventually define language as follows: '[a] *language* is any set of sentences over an alphabet. Most languages of interest will contain an infinite number of sentences' (1). Within the linearized space of the alphabet the possibility of infinity is allowed: symbols can be infinitely re-combined, but only in linear terms, as sequences. Moreover, the length of a string is finite (there is no infinite string), but there can be infinite strings, or infinite linear re-combinations of discrete symbols. This is a kind of infinity which is kept within the limits of linearization.[23]

At this point, Hopcroft and Ullman raise the important question of how to 'represent (i.e., specify the sentences of) a language' (1). They write:

> If the language contains only a finite number of sentences, the answer is easy. One simply lists the finite set of sentences. On the other hand, if the language is infinite, we are faced with the problem of finding a finite representation for the language. The finite representation will itself usually be a string of symbols over some alphabet, together with some understood interpretation which associates a particular representation to a given language.
>
> (Hopcroft and Ullman 1969: 1f.)

First of all, it must be noticed here that the 'representation' of a language is defined in terms of the 'specification' of its sentences. In this way, representing a language amounts to *writing* down all its sentences. This kind of 'representation' is therefore an *inscription*. In an analogous way, as I showed in Chapter Three, the participants in the NATO Conferences on Software Engineering thought that specifying a software system amounted to repeatedly inscribing it - that is, writing and rewriting a description of it until this description 'became' the system itself. However, as Hopcroft and Ullman remark, listing all the sentences of a language works well for finite languages: in other words, languages that have a finite number of sentences

[23] Jay Bolter remarks in his book of 1984, *The Turing Man*, that computers do not deal with infinity. Being material, they can only deal with finite representations of numbers (Bolter 1984). I will show in the rest of the chapter how this is not always the case. For instance, a computer can go into an infinite loop and keep performing the same operations over and over – at least until the computer crashes or is rebooted.

are linearized in the form of a list. But infinity raises a problem. Infinite languages require a different kind of inscription, which must be capable of dealing with infinity while remaining itself finite (in fact, representation can also be considered a string, and a string is finite by definition). However, for Hopcroft and Ullman the representation of an infinite language is not just a string, or a sequence over some alphabet, but a string with something else attached – namely, what they call 'an understood interpretation' that associates the representation with the language represented. Here, Hopcroft and Ullman's references to 'understanding' and 'interpretation' seem to hint at a concept of language as a common agreement between speakers over some way of conveying meaning, or over the idea of 'communication' as it is traditionally formulated (Derrida 1988). However, I will show in a moment that such 'understood interpretation' that links language and representation is nothing but a 'grammar'. In other words, it is just another material inscription of what the authors call 'language'. Rather than resort to the concept of shared meaning, they actually devise a very clever mode for re-inscribing formal languages. But what are the characteristics of this particular inscription they call 'grammar'?

First of all, let me briefly explain how Hopcroft and Ullman contextualize the issue of language description, since grammars are just *one* of the possible methods for describing languages, albeit the most commonly used one for programming languages. Languages, Hopcroft and Ullman write, can be represented from two different points of view: the 'recognition point of view' and the 'generative point of view' (5). More precisely,

> [o]ne way to represent a language is to give an algorithm which determines if a sentence is in the language or not. A more general way is to give a procedure which halts with the answer 'yes' for sentences in the language and either does not terminate or else halts with the answer 'no' for sentences not in the language. Such a procedure or algorithm is said to *recognize* the language.
>
> (Hopcroft and Ullman 1969: 5)

It is not important here to enter into the details of Hopcroft and Ullman's distinction between an algorithm and a procedure (two terms they also happen to deploy as synonyms from time to time). What is more striking is that, according to the above passage, the first way of specifying a language is by *giving an algorithm* for the *recognition* of sentences belonging to the language itself. One must be reminded at this point that an algorithm is 'a finite sequence of instructions that can be mechanically carried out, such as a computer program' (2). Thus, an algorithm is itself a linear construct. Once again, we are brought back to the linearization of time: the algorithm is defined as a sequence; it is a linear structure, sequenced in time as a chain of actions/moments. Not only are we reminded again of Leroi-Gourhan's *chaine operatoire*, but, even more importantly, the linearization of time is revealed to permeate every aspect of software development. As I showed in Chapter Three, it also constitutes the (constantly undone and constantly reinstated) foundation of Software Engineering as the linearization of the time of software development. Following Leroi-Gourhan and Derrida, the phonetic alphabet is a linearization of graphism according to the time of sound. For Hopcroft and Ullman, the process of language recognition also requires the linearization of time – although the languages they are dealing with are not (primarily) phonetic. I will show in the rest of this chapter how time is linearized in code - that is, how time needs to be *made linear* in order for code to exists and how, nevertheless, such linearization is never complete.

In fact, in the above passage Hopcroft and Ullman define the representation - that is, the specification, or inscription - of a language in terms of an algorithm. Therefore, the representation of a language equals the method for including/excluding sentences from that language, or the method for constructing the boundaries of the language. Moreover, if this can be done in an automated way, not only have we got rules for including/excluding sentences, but we actually have an autonomous mechanism that operates the process of making a choice between the inclusion/exclusion alternatives. In other words, the algorithm for recognizing a language is a sequence of choices between sequences (strings) of alphabetical symbols. Language is the result of chained choices operated in the linearized realm of an alphabet. It looks like one starts by generating all the possible sequences over the given alphabet: one starts from pre-existent, previously formed sequences, and

then the algorithm 'recognizes' (or not) each of those sentences. But what do Hopcroft and Ullman mean by 'recognition'? What does recognition actually *do*? Hopcroft and Ullman define recognition as the halting of a procedure and as the return of a binary answer. Ultimately, then, recognition is either a '0' or a '1'. The binary (alphabetical) outcome of a sequence of steps determines the belonging of a sequence of alphabetic symbols to the realm of the 'properly linearized' sequences of symbols that constitute a language. Language could hardly be more linearized than this.

But a language can also be represented from a generative point of view. In other words, a procedure can be defined which 'systematically generates successive sentences of the language in some order' (5). To explain this point better, the authors treat the problem of generating a language over an alphabet V formally. The general idea is that

> [i]f we can recognize the sentences of a language over alphabet V with either an algorithm or a procedure, then we can generate the language, since we can systematically generate all sentences in V*, test each sentence to see if it is in the language, and output in a list only those sentences in the language.
>
> (Hopcroft and Ullman 1969: 5)

Again, it is not so important to understand the distinction between algorithm and procedure. The relevant point of Hopcroft and Ullman's argument is that the final result of this second approach is still a *list* – albeit one inscribed through mechanical steps. Moreover, this passage links the process of recognition to the one of generation by exercising such recognition over (and *after*) the generation of *all* the possible strings over a given alphabet. Hopcroft and Ullman specify that in this second case the recognizing procedure must be designed in such a way that it always halts at some point in time. This is not banal, since here 'procedure' is intended as a computer program, and, as we have seen above, some computer programs (generally ill-designed ones) can never come to a halt. This is the case,

for example, for loops that never fulfil their halting conditions.[24] A recognizing procedure that does not halt could go on analyzing the same sentence forever, and it would never generate a list that describes the language. However, the main point I want to highlight here is that, although an algorithm is sequenced in time, a major dilemma with regard to time and language is evidenced in Hopcroft and Ullman's passage. On the one hand, a sentence belongs to a language if (and only if) it belongs to the set of sentences that constitute that language. On the other hand, it is precisely this process of choice - that is, of exclusion - that sets up the boundaries of that set. In other words, we could ask: does a set pre-exist the construction of its boundaries? Do sentences pre-exist their belonging to a language? To preserve the sequencing in time of the algorithm, the authors try to set up and maintain a temporal distinction between generating all the possible sentences and *then* determining which ones are correct. Linearization is here presented as a two-step process of indiscriminate sequencing followed by choice (with the correspondent exclusion or discarding of the 'wrong' sentences).

To sum up the argument so far, we have seen how for Hopcroft and Ullman a programming language is always a formal language. In turn, a formal language is a (generally infinite) set of sequences of symbols 'over an alphabet'. To specify any formal language, a set of rules must be defined to which the sentences of that language must conform in order to be part of the language. These rules can be embodied in an automatic sequence of steps which decides which sentences are correct and which rules out the incorrect ones. This automatic sequence of steps defines the boundaries of the language, and it can well be a computer program itself. Therefore, there are computer programs whose purpose is to analyse and/or generate formal languages. Moreover, the strings that these computer programs analyse can themselves be computer programs. In this case, a computer program which carries out the analysis of other computer programs is called a 'compiler'.[25] Being a computer program, the compiler is itself written in a programming language, which in turn has been specified and (typically) compiled.

[24] As I pointed out both in Chapter One and earlier on in this chapter, circular time is linear too (Derrida 1976).
[25] It can actually be either an interpreter or a compiler, although for the purposes of this chapter it is not important to enter into so much technical detail. I will return to compilers later on in this chapter.

In the further part of the book, Hopcroft and Ullman concentrate on generative systems for the representation of languages. For them, the generative systems that are of primary interest for the description of programming languages are the 'systems known as grammars' (8). This means that grammars are a subset of all the possible ways in which languages can be linearized. The notion of grammar, the authors acknowledge, comes from linguistics. They write:

> The concept of a grammar was originally formalized by linguists in their study of natural languages. Linguists were concerned not only with defining precisely what is or is not a valid sentence of a language, but also with providing structural depictions of the sentences. One of their goals was to develop a formal grammar capable of describing English.
>
> (Hopcroft and Ullman 1969: 8).

Quite obviously Hopcroft and Ullman refer to Chomsky's theory here, and explicitly to his famous article of 1956 entitled 'Three Models for the Description of Language'. Again, they remark that at the early days of generative linguistics '[i]t was hoped that if, for example, one had a formal grammar to describe the English language, one could use the computer in ways that require it to "understand" English. Such a use might be language translation or the computer solution of word problems' (8). They also acknowledge that, as of 1969, this goal is still basically unrealized:

> We still do not have a definitive grammar for English, and there is even disagreement as to what types of formal grammar are capable of describing English. However, in describing computer languages, better results have been achieved. For example, the Backus Normal Form used to describe ALGOL is a 'context-free grammar', a type of grammar with which we shall deal.
>
> (Hopcroft and Ullman 1969: 8f.).[26]

[26] I will return to the Backus Naur Form (BNF) in a moment (and we shall see how Hopcroft and Ullman's formal notation for a grammar is actually modelled on the BNF). As I have mentioned at the beginning of this chapter, the ALGOL programming language was defined between the late 1950s and early 1960s and was mainly used in scientific applications.

I have already commented upon the fact that it is really not surprising that the best results with formal grammars have been obtained in the description of programming languages, since programming languages have been constructed according to formal grammars. It is also important to notice at this point that the linearization of programming languages entails what Clark (2000) has called the 'complicity of technology with metaphysics'. In other words, software is complicit with the way in which linearization has operated so far in Western thought, especially in linguistics. It could also be said that, in the theory of programming languages, the complicity of software with metaphysics takes the form of a complicity with structural (Chomskyian) linguistics. In the non-phonetic realm of programming languages linearization functions as the discretization and sequentialization of computation into alphabets and grammars. The time of this linearization is not related to sound (it is not the time of the pronunciation of speech). Rather, it is the time of software compilation and execution by the computer (and hence is ultimately regulated by the computer's internal clock), as I will show later on in this chapter. However, to understand how grammars work let me now go a little deeper into Hopcroft and Ullman's explanation. They analyze an English sentence to explain the basic concepts of the 'tree-diagram' and of the 'rules of production'.[27] They write:

> For example, the sentence 'The little boy ran quickly' is parsed by noting that the sentence consists of the noun phrase 'The little boy' followed by the verb phrase 'ran quickly'. The noun phrase is then broken down into the singular noun 'boy' modified by the two adjectives 'The' and 'little'. The verb phrase is broken down into the singular verb 'ran' modified by the adverb 'quickly'. ... We recognize the sentence structure as being grammatically correct.
>
> (Hopcroft and Ullman 1969: 9)

Hopcroft and Ullman start out by 'parsing' the given sentence. 'Parsing' is a term that has become common in the theory of programming languages and – as will

[27] The authors' choice of example is very telling, since, although the best results have been obtained with programming languages, it is still intuitively easier to use a 'natural language' as an example. I am following Hopcroft and Ullman here; later on I will give an example based on an actual programming language.

become clear further on – of compilers, and it broadly means 'breaking down into components'. The sentence structure is then diagrammed in the form of a tree-diagram. Hopcroft and Ullman's process of parsing is meant to prove the *correctness* of the analysed sentence.[28] It must be noted at this point that the technical literature on programming languages distinguishes very carefully between syntactic and semantic correctness – namely, between how a sentence is constructed and what its meaning is. Generally speaking, a computer program is syntactically correct when all its parts are well formed according to the rules of that language; it is semantically correct when it does what it is meant to do when executed. For the moment, I will concentrate on syntactic correctness, since this is what formal grammars check. I will investigate semantic correctness later on in this chapter.[29]

The process of parsing, being a process of 'breaking down' a given string, consists in looking for blanks or word separators (and it must be kept in mind that, when viewing the sentence as one string, the blanks are just part of that string, or symbols in the alphabet) and in decomposing the string into sub-strings according to certain rules. Hopcroft and Ullman explain:

> The rules we applied to parsing the above sentence can be written in the following form:
>
> <sentence> → <noun phrase> <verb phrase>
> <noun phrase> → <adjective> <noun phrase>
> <noun phrase> → <adjective> <singular noun>
> <verb phrase> → <singular verb> <adverb>
> <adjective> → The
> <adjective> → little
> <singular noun> → boy
> <singular verb> → ran

[28] Although Hopcroft and Ullman's grammar is meant as a generative instrument, it can also be used for checking the correctness of given sentences. It will therefore come as no surprise that this grammar also constitutes the basis for compilers – namely, for devices that analyse a program (already written by a programmer in a high-level language) and check it for correctness by decomposing it according to predefined grammatical rules.

[29] More precisely, I will show that three kinds of correctness are analysed by compilers, the third one being 'lexical' correctness - that is, the check (which is in fact carried out as a first step) that all the symbols of a given string belong to the given alphabet.

<adverb> → quickly

> The arrow in the above rules indicates that the item to the left of the arrow can generate the items to the right of the arrow.
>
> (Hopcroft and Ullman 1969: 9)

In this (Chomskyan) formalism the names of the parts of the given sentence (such as 'noun', 'verb', 'verb phrase', etc.) are enclosed in brackets. Here, again, I want to stress how the distinction between the stages of recognition and generation is reaffirmed while at the same time being revealed as rather unstable. The authors start from a given English sentence and decompose it in order to show how it can be recognized as a well-formed string in the English language. However, when outlining the rules for recognition (in fact, the rules without which the sentence could not possibly be broken down in the first place), they write them in the form of rules of generation. In other words, a sentence cannot be analysed without us already knowing how it should be formed – and it cannot be formed without knowing how it would be analysed. This is probably the reason why the best results with formal grammars have been achieved with programming languages, which have been constructed according to such formal grammars.

However, the rules guiding the process of parsing/generation appear as rules of replacement (or substitution) of one string with another: the process is explained as a re-inscription of some bracketed string into one (or more) bracketed string or one (or more) non-bracketed string. This re-inscription of strings is governed by the arrow (the typical operator of the 'rules of production'). The arrow indicates a movement by which a certain inscription is replaced by another one within the discrete and ordered space of the line. Hopcroft and Ullman further separate the parsing of a string from its generation in the following way:

> One should note that we cannot only test sentences for their grammatical correctness, but can also generate grammatically correct sentences by starting with the quantity <sentence> and replacing <sentence> by <noun phrase> followed by <verb phrase>. Next we select one of the two rules for <noun phrase> and apply it, and so on, until no further

> application of the rules is possible. In this way any one of an infinite number of sentences can be derived – that is, any sentence consisting of a string of occurrences of 'the' and 'little' followed by 'boy ran quickly' such as 'the little boy ran quickly' can be generated. Most of the sentences do not make sense but, nevertheless, are grammatically correct in a broad sense.
>
> (Hopcroft and Ullman 1969: 9)

It is quite clear from the above passage that the notion of temporality becomes of fundamental importance in the (unstable) distinction between parsing and generation. The main point of this long and complex quote is that the authors deal with temporality as a process of substitution, which is also a process of re-inscription. Parsing and generating sentences are described as two processes governed by a different direction in (linear) time. Parsing is described as a breaking down of the string, while generation is a replacement of strings. The temporal dimension of the process of 'generation' is clear from the above passage: 'next' explicitly identifies the process of generation as a sequence of choices of replacement, 'until no further replacement is possible'. Here linearization *is* a direction in time. Similarly, the term 'occurrence' hints at the fact that the time of the string is a linear time. Importantly, the closure ('until no further application of the rules is possible') clearly indicates that the string has to come to an end, or to a point and moment on the line of substitutions when and where it terminates. At the same time, though, the generative grammar thus defined is also a means of 'enabling' infinity – that is, the possibility of generating infinite strings, each of finite length. Ultimately, the time of linearization is both constituted and destabilized by the distinction between recognition and generation. Generation is always already recognition, but it is necessary to maintain the distinction between the two in order to make it possible to write a grammar – that is, a set of rules of re-inscription (or substitution) governed by an arrow (orientated in time and space). Re-inscription is therefore constituted together with the linearization of time – even though this is not the time of speech, but the time of code compilation and (as I will show in a moment) execution. Let me now investigate a little further how the process of material re-inscription works in Hopcroft and Ullman's rules of production.

As Hopcroft and Ullman explain, the strings enclosed in angular brackets, such as <singular noun>, <verb phrase>, <sentence>, from which strings of words can be derived, are called 'syntactic categories', 'nonterminals', or 'variables'. The strings which are not enclosed in angular brackets - that is, as the authors explain with characteristic awkwardness caused by the self-reflexivity of their meta-language, 'the objects which play the role of words' - are called 'terminals'.[30] The relation that exists between strings of variables and terminals (that are indicated by the arrow) are called 'productions'. Examples of productions are <noun phrase> → <adjective> <noun phrase> or <singular noun> <singular predicate>→ <singular noun> <adverb> <singular verb>. A production that has the form <noun phrase> → <adjective> <noun phrase> can generate infinite sentences because it allows the syntactic category <noun phrase> as a possible substitution for itself. Such a rule is said to be 'recursive'. Recursivity makes it possible to generate (and therefore, to specify) an infinite number of strings through a finite number of rules. This is exactly the goal that Hopcroft and Ullman set themselves at the beginning - to find a finite specification for an infinite language. I want to point out here that recursivity allows infinity to emerge from finity – in other words, one can keep applying recursivity indefinitely, thus generating infinity. This is how, as I have pointed out earlier on in this chapter, Hayles understands complexity as being generated in computation through iteration. In fact, I want to emphasize here that iteration is a really common mode in which computation functions – which nevertheless does not seem to me to set computation apart from 'language' or

[30] A natural language can be deployed to explain itself (or to reflect on itself) or another language, and this use of the language is called 'metalinguistic'. For instance, during a Spanish grammar lesson, we can use the English language to explain Spanish – therefore, we use the English language as a meta-language. Similarly, a textbook of C language can use English language to explain C, again using English as a meta-language. This capacity for speaking about itself is generally considered typical of natural languages. However, self-reflexivity can cause 'antinomies', which result from a self-reflexive use of a language that, while attempting to explain the language, 'disturbs' the language itself. The need for a universal meta-language has been felt by the scientific community for centuries, and Chomsky's generative grammar can be seen as a partial attempt in this sense. Famously, Gödel has provided a mathematical proof that, within a formal system, there are always propositions that cannot be proved true or false on the basis of that system's foundations (axioms). Thus, a meta-language capable of solving every mathematical controversy and ambiguity, therefore allowing for a perfectly consistent construction of mathematical knowledge, cannot actually exist. After Gödel, the scientific community knows that only languages 'reasonably' (but not totally) consistent exist. It might perhaps be said that Gödel's theorem hints at the fact that linearization, when it aims at being complete (or universal), becomes undone – in other words, he acknowledges that the process of linearization must always be incomplete, or leave something outside itself, in order to exist. No matter how rigorous and extended a linear system is, there will always be something that escapes its linearity.

'writing' the way Hayles sees it. However, Hopcroft and Ullman remark once more that, although an infinite number of sentences can be generated by applying some rules indefinitely, any single string must stop at some point, therefore being finite. Actually, they speak here of strings as 'quantities', thus clearly showing that a formal grammar is intended to work on a computer.[31] We shall see throughout the rest of this chapter how computation is performed through a process of replacement, or re-inscription. To be more precise, I will show how re-inscription is the point at which linearization is exceeded and at which the unforeseen consequences of technology are generated.

In order to 'formalize the notion of grammar', Hopcroft and Ullman propose a specific notation for grammars. Besides the definitions of nonterminals, terminals, and productions, another definition is given, that of the 'start symbol' – which is one nonterminal that 'generates exactly those strings of terminals that are deemed in the language' (10). There is only one start symbol per language. In Hopcroft and Ullman's example, <sentence> is the start symbol. As Aho et al. (2007) point out, '[u]nless stated otherwise, the head of the first production is the start symbol' (199). I want to emphasize here that, with the start symbol, a point of origin is established for the language. Actually, a language consists of all the strings of terminal symbols that can be generated *by* the start symbol.

Hopcroft and Ullman finally provide the following formal *notation* for a grammar: 'we denote a *grammar* G by (V_N, V_T, P, S). The symbols V_N, V_T, P, and S are, respectively, the *variables*, *terminals*, *productions*, and *start symbol*. V_N, V_T, and P are finite sets'. Furthermore, 'the set of productions P consists of expressions of the form $\alpha \rightarrow \beta$, where α is a string in V^+ and β is a string in V^*. Finally, S is always a symbol in V_N' (10).[32] This definition makes very clear once again the written nature of grammars. In fact, terminal symbols are lowercase letters from the beginning of the alphabet (such as *a, b, c*); operator symbols such as +, *, and so on; punctuation symbols such as parentheses, commas, and so on; digits from 0 to 9, and (as a later

[31] Sentences need to be finite because, as I have pointed out earlier on, computers can only deal with finite quantities.

[32] It must be noticed that, to define grammar G, the authors have used English as a meta-language. It must also be noted that Hopcroft and Ullman's notation for a grammar coincides with the Bakus Naur Form (BNF).

text will add with specific reference to the generation of programming languages) 'boldface strings such as **id** or **if**, each of which represents a single terminal symbol' (Aho et al. 2007: 198). Nonterminals are uppercase letters which occur early in the alphabet (such as A, B, C), the letter S (which, when used, is usually the start symbol), and 'lowercase, italic names such as *expr* or *stmt'* (198). Besides, and even more importantly, the process of replacement described by the rules of production, and by the branching out of the tree-diagram, is presented here not only as an orientated linearized movement, but also as a movement *from left to right* that follows the Western movement of the eyes of the reader – a reader, that is, of alphabetical writing. This has even more interesting implications when we consider the tree-diagram. In a subsequent passage, Hopcroft and Ullman define what they call the 'derivation tree for context-free grammars' as follows:

> We now consider a visual method of describing any derivation in a context-free grammar. A *tree* is a finite set of *nodes* connected by directed *edges*, which satisfy the following three conditions (if an edge is directed from node 1 to node 2, we say the edge *leaves* node 1 and *enters* node 2):
>
> 1. There is exactly one node which no edge enters. This node is called the *root.*
> 2. For each node in the tree there exists a sequence of directed edges from the root to the node. Thus the tree is connected.
> 3. Exactly one edge enters every node except the root. As a consequence, there are no loops in the tree.
>
> (Hopcroft and Ullman 1969: 18f.)

This visual representation of linearity in/as a tree deserves careful analysis.[33] Movement and time are also part of the tree. There are departures and arrivals: edges have one and only one direction; every edge describes one orientated movement; an edge 'leaves' one 'node' and 'enters' another. It must be pointed out that the tree reaffirms, first and foremost, the existence of a point of *origin.* The

[33] *Pace* Alexander Galloway and his (Deleuzian) affirmation that 'we are tired of trees', in fact trees are still used a lot in textbooks regarding programming languages - and in general in computer science (Galloway 2004: 24).

origin is defined by it being un-originated, and visually by its *impenetrability*: no edges enter it, and it is the only un-entered node. The process of linearization here establishes a point of origin for itself. Such a point of origin is nominated as a 'root' - where the metaphors of kinship and genealogy actually seem predominant over the metaphors of biology, since the tree grows upside down, as Jay Bolter (1984) once noticed in relation to the many trees of computer science. Importantly, all the nodes are connected *to the root*. There is no stand-alone node in the tree; moreover, there is no node unconnected to the origin. Although the edges are orientated from departure to arrival, every node is traceable backwards; every node has an origin, which is the same for all the other nodes. We can always go backward; time can always be reverted to its starting point. To make traceability possible, the movement throughout the tree needs to be controlled. Therefore, the third point listed by the authors is also very important, since it excludes loops from the linearized space of the tree. Not only is the tree linear; it is also *loop-free*. This is achieved by handling nodes so that every node allows only one edge to enter it, with the origin still being the exception. In fact, while reversals, or returns to the origin, are possible, what is impossible is entering a reiteration that causes infinity. In sum, a certain kind of repetition is made impossible – namely, the repetition that causes infinite loops. Repetition (at least, the 'dangerous' kind of repetition) is formally expelled from the tree – although, as I have already pointed out, repetition is what constitutes the mark – and thus also the possibility of symbols, alphabets and trees. Similarly, in Chapters Three and Four I showed how the *loop* constitutes a problem in software design, while at the same time making software design possible: reiteration (the repeated attempt at constructing the software system) is both what makes the system possible and what threatens it with the impossibility of ever being finished, or with producing unexpected consequences as unforeseeable variations through repetition. What I want to emphasize here is that iteration is always un-manageable in software *at each level*, from the tree-diagram of context-free grammars to the broader process of software development. However, since iteration is also unavoidable, its expulsion from software is impossible. In fact, iteration – i.e., the process of generating differences in repetition – is a constitutive characteristic of software, just as fallibility (and the capacity for generating unforeseen consequences) is constitutive of technology. The expulsion of the loop from the tree corresponds to the expulsion of risk from technology. But risk is

always implicit in technology. In fact, the definition of iteration as difference in time makes re-inscription always risky, because it is always open to variation. I will return to this important point in a moment. For now, let me explore the relationship between the string and the tree in order to show how the process of re-inscription links the two. Hopcroft and Ullman give the following definition:

> The set of all nodes n, such that there is an edge leaving a given node m and entering n, is called the set of *direct descendants* of m. A node n is called a descendant of node m if there is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that $n_k = n$, $n_1 = m$, and for each i, n_{i+1} is a direct descendant of n_i. We shall, by convention, say that a node is a descendant of itself.
>
> (Hopcroft and Ullman 1969: 19)

Although here the term 'descendants' makes the kinship metaphor even more clear, the most important point in this passage is the direction of the edges. Such direction, in turn, determines how a given node is classified and how the re-inscription of the tree as a string is performed. Hopcroft and Ullman continue:

> Some of the nodes in any tree have no descendants. These nodes we shall call *leaves*. Given any two leaves, one is to the left of the other, and it is easy to tell which is which. Simply backtrack along the edges of the tree, toward the root, from each of the two leaves, until the first node of which both leaves are descendants is found. If we read the labels of the leaves from left to right, we have a sentential form. We call this string the *result* of the derivation tree.
>
> (Hopcroft and Ullman 1969: 20)

This description of the passage from tree to string is extremely significant. This passage from the two-dimensional space of the tree to the one-dimensional string is realized through an orientated movement either from the left to the right or from the right to the left. The space of the tree is therefore an orientated space. The most important aspect here is that it is possible to distinguish the left from the right *because* the left and the right actually coincide with the left and the right of the

written page (that is, the left and the right related to print, to the book, to traditional Western writing). Thus, the passage from the tree-diagram to the string that is characteristic of formal grammars can only exist *because* there is a consolidated tradition that establishes what a page is, what the left and the right of the page are, what it means to read, and in what direction one reads. In this sense formal grammars, the programming languages which are based on formal grammars and programs written in those programming languages can all exist *because* of the page.[34]

Even more importantly, however, the re-inscription of the strings of a language as rules of production or as a tree-diagram is made possible by the presence of the graphic interruption within strings. In Hopcroft and Ullman's example, the interruption, understood as the separation between English words, is what enables both the transcription of the given English string *as* a set of rules of production, and the transcription of the rules of production *as* a set of English strings. The same can be said for programming languages. For instance:

stmt → **if** (*expr*) *stmt* **else** *stmt*

The above rule of production establishes that an if-else statement in the Java programming language is the concatenation of the keyword **if**, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword **else** and another statement. In this rule of production, **if** and **else** are terminals, while *stmt* (statement) and *expr* (expression) are nonterminals (defined by other rules of

[34] Importantly, the authors state that 'a derivation tree is a very natural description of a particular sentential form of the grammar G' (Hopcroft and Ullman 1969: 20), therefore claiming that a tree form is essentially more 'natural' than a sentential form, and actually evaluating these two forms of linearization according to a presupposed 'naturalness'.

production). The above rule of production is expressed using the Backus Naur Form.[35]

In order to understand better the role played by the symbols of interruption (such as the semicolon or the blank) in programming languages, it is worth now examining in detail the concept of the 'compiler'. Although the main body of Hopcroft and Ullman's book is devoted to the relation between grammars and automata, and although there is a straightforward relationship between automata and compilers, they do not give a discursive account of what a compiler is. As we have seen, an automaton is a formalism used to describe systems and their transformations (Aho et al. 2007: 147). A specific class of automata, named 'finite automata', do exactly what Hopcroft and Ullman have described above: they recognize the sentences belonging to a language. In other words, finite automata are *recognizers*: they give a 'yes or no' answer regarding each possible string, depending on the compliance of that string with the rules of a given grammar. Compilers are finite automata. Let me now take a little detour from Hopcroft and Ullman's book in order to explain how compilers work and why they are crucial in my argument.

To understand the role and importance of compilers, it must first be noticed that, during the phase of the implementation of a software system after programs have been written, the implementation itself is not really finished. What has been written is generally a large ensemble of interconnected programs, expressed in some high-level programming language. All this, as argued in Chapters Three and Four, forms

[35] The BNF is a widely used notation for the description of grammars – and specifically for the so-called context-free grammars. In fact, different types of grammars can be defined. Within the scope of this chapter, suffice it to say that some grammars can be very powerful and complex, and therefore able to describe complex languages, but they are not very easy to understand or use. Others can be easier to use but less powerful. Chomsky (1956) provided a classification of grammars on the basis of their complexity: he distinguished grammars of 'type 0', '1', '2', and '3'. Again, for the purposes of this chapter the most relevant difference is the one between context-sensitive grammars and context-free grammars. In context-free grammars, a string can substitute for another string whatever the context. Conversely, in context-sensitive grammars a string can be replaced by another one only when the former is part of a certain context. A typical example is the use of the adjective 'his/her' in English, which must be chosen according to the grammatical gender of the noun to which it refers. The kind of grammar used to describe programming languages is context-free, because programming languages do not have all the restrictions that 'natural' languages have. The BNF was presented by John Backus at the first World Computer Congress in Paris in 1959 as a formal description of what would later become the ALGOL programming language. Later on Peter Naur proposed the name 'Backus Normal Form' for such notation, and he worked on it in order to simplify it. His name was then added to the name of the grammar in recognition of his contribution - the Backus Normal Form thus becoming known as the Backus Naur Form.

the *state* of the software system at this point. For instance (assuming that the system is written in Java language), a very tiny program fragment might have the following form:

```
int x; int y; int z;

{
x=x+y;
}
```

This example makes clear that a program is inscribed in a form that takes into account the space of the page (even if the page is visualized on a screen) and that respects the conventional direction of (Western) reading. However, from the point of view of the theory of formal languages the program is one continuous string, with blanks and semicolons coded as ASCII characters that are also part of the string. In fact, the use of new lines and tabulations has the only function of making the program more easily readable for the human eye. In order to be re-inscribed *as* circuits (another *state* of software), this program needs to undergo not just one but many re-inscriptions. As Aho et al. remark in the 'purple book', '[b]efore a program can run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called *compilers'* (Aho et al. 2007: 1). For the sake of clarity, let me now follow Aho et al. (2007) in their informal and intuitive explanation of compilers. I will come back to Hopcroft and Ullman's text at the end of this section. Also, while I am examining here the description of a compiler given by Aho et al. (2007), it must be kept in mind that I am not focusing on the actual functioning of the compiler, but rather on the transformation undergone by the originary program (the source program), in order to show how the process of its re-inscription works.

Aho et al. (2007) define a compiler as follows:

> Simply stated, a compiler is a program that can read a program in one language – the source language – and translate it into an equivalent

> program in another language – the target language. … An important role of the compiler is to report any errors in the source program that it detects during the translation process.
>
> (Aho et al. 2007: 1)

A compiler is thus, to use Hopcroft and Ullman's term, an automaton that 'reads' any program previously written in the high-level programming language for which the compiler has been designed (the 'source language') and re-inscribes it as binary code (the 'target language'), or, as I will show in a moment, *as* circuits. The terms 'read' and 'translates' used in the above passage are metaphors for the material re-inscription of the source program – a process of re-inscription that is technically called 'compilation'. But how does this re-inscription work? What does it *do*?

The authors establish that the process of compilation can be broken down into different phases. They write:

> The first phase of a compiler is called *lexical analysis* or *scanning*. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token … that it passes on to the subsequent phase, syntax analysis.
>
> (Aho et al. 2007: 5f.)

This passage uses all the rhetorical devices that are common in technical descriptions of programs: it switches from terms denoting temporal 'phases' of execution of the program ('lexical analysis') to terms denoting agency of (parts of) the program itself ('lexical analyzer'). Yet, what this quote makes clear is that the process of compilation consists first of all of detecting 'interruptions' in the string of the program (such as blanks and semicolons) and of re-grouping the symbols of the string into smaller sub-strings called 'lexemes'.[36] It must be noted that the compiler is an automaton that follows the rules of production of a specific grammar

[36] The 'blank' is understood here not as Derrida's 'mark', but as a specific binary code for a blank space in the page. This means that a blank is physically inscribed into the memory of the computer and it occupies the physical space of (usually) two bytes.

– and it is this grammar that establishes what works as an interruption in the source program (for instance, a blank space or a ';' work as interruptions, but not an 'a'). The symbol of interruption therefore ruptures the chain of symbols and functions as a discriminating tool for re-grouping the other alphabetical symbols into new groups ('lexemes'). For each lexeme, the phase of lexical analysis produces a 'token' – in other words, each lexeme is re-inscribed as a slightly different string with a few explanatory symbols added. At this point, the initial single string of the source program has been completely re-inscribed. I want to emphasize here how the lexical analyzer also works on the basis of linearization and discreteness: its function is to 'read' the ruptures in the source program, to make them work as ruptures, to make them function in the context of the computer so that the originary string is transformed into a different string (the sequence of tokens). The lexical analyzer *works with the rupture* in order to re-inscribe the string. The ruptures (the blanks, the semicolons) disappear from the initial string *as symbols* - that is, as characters with their own binary encoding in the computer memory - and they are re-inscribed literally *as* the 'tokenization' of the program. They are actually what enables such tokenization.

Starting from the sequence of tokens produced in the phase of lexical analysis, another re-inscription is carried out. The authors continue: 'The second phase of the compiler is *syntax analysis* or *parsing*. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream' (Aho et al. 2007: 8). Again, it is not important to understand all the technicalities of this passage. What is important is that the tokens are 'rearranged' once more according to the syntactical rules of the grammar on which the language is based, with the help of the additional symbols inscribed into the tokens in the previous phase. The way this happens varies physically: obviously no graphical representation of a tree is actually 'depicted' in the memory of the computer. The parse tree is also a rebuilt string: in other words, the sequence of tokens is regrouped in order to be then analyzed by the 'semantic analyzer', which in turn checks the consistency between the tokens and produces the so-called 'intermediate code'. Aho et al. (2007) explain: 'In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety

of forms' (Aho et al. 2007: 9). They define the intermediate code as 'assembly-like', and we shall see in a moment what such intermediate code looks like. The intermediate code is a further re-inscription of the source code, a further 'externalization' of the software system, and a further transformation of the ruptures inscribed in the source program - ruptures that actually make it possible for code to function.

It is important to notice at this point that, while performing all these re-inscriptions of the source program, a compiler also checks the program for correctness – that is, it assesses the program's compliance with the rules of transformation defined for the specific programming language in which it was written (Aho et al. 2007: 1). However, a compiler can only detect errors according to such rules (for instance, it can easily locate a missing ';') but it cannot anticipate whether the source program will function in an unpredictable way when executed. For instance, it cannot anticipate whether a perfectly correct portion of the source program will cause the computer to enter an infinite loop with unforeseeable consequences. Let me provide a further example.[37] One of the most famous bugs in software – so famous that in the late 1980s professionals started wondering whether it was in fact just an urban myth – is the following FORTRAN statement:

```
DO 10 I = 1.10
...
10 CONTINUE
```

This legendary piece of code is largely quoted by technical literature as well as by the many websites devoted to the discussion of software errors.[38] When executed, it leads to an infinite loop, which erroneously has been held responsible for the failed launch of Mariner I, the first American space probe, in 1962. In a posting dated 28

[37] Obviously the history of Software Engineering has seen many attempts at improving the capability of compilers to detect errors in source programs, as well as the general capacity of the phase of testing for revealing malfunctions before the system is released. However, I am not interested here in the technical ways of improving compilers' technical characteristics, but rather in the uneliminable capacity of technology for generating the unexpected.

[38] As I explained in Chapter Four, a bug is an error in software that is not easily detectable. The etymology of the term hints at the fact that bugs are often very hard to find – as was the moth which was found trapped in a relay of the electromechanical computer Mark II in 1945, and which caused many malfunctions.

Feb 1995 on http://www.rchrd.com/Misc-Texts/Famous_Fortran_Errors John A. Turner debunks the myth of the FORTRAN error by referring to a previous discussion posted by Dan Pop on the newsgroup http://comp.lang.fortran on Mon, 19 Sep 1994.[39] To give but one example of the many pieces of narrative circulating on the Internet in relation to this bug, Turner mentions how one of the Frequently Asked Questions on http://alt.folklore.computers is 'III.1 - I heard that one of the NASA space probes went off course and had to be destroyed because of a typo in a FORTRAN DO loop. Is there any truth to this rumor?'. Also, Dieter Britz is reported to have posted the following question on http://comp.lang.fortran: '[i]t is said that this error was made in a space program program, and led to a rocket crash. Is this factual, or is this an urban myth in the computer world?'. Other incorrect (but often sharp and colorful) versions circulate, such as those posted by craig@umcp-cs ('[t]he most famous bug I've ever heard of was in the program which calculated the orbit for an early Mariner flight to Venus. Someone changed a + to a - in a Fortran program, and the spacecraft went so wildly off course that it had to be destroyed') and by Sibert@MIT-MULTICS ('it is said that the first Mariner space probe, Mariner 1, ended up in the Atlantic instead of around Venus because someone omitted a comma in a guidance program'), both reported by Dave Curry in his piece on 'Famous Bugs' on http://www.textfiles.com/100/famous.bug.[40]

The legend results from the confusion of two separate events: Mariner I was in fact destroyed on 22 July 1962, when it started behaving erratically four minutes after launch, but such an erratical behaviour was due to a combination of hardware and software failures (Ceruzzi 1989: 202 f.), while the DO-loop piece of code was identified (and corrected) at NASA during the summer of 1963 during the testing of a computer system. In fact, according to the story originally recounted by Fred Webb on http://alt.folklore.computers in 1990 and subsequently reported by Turner, during the summer of 1963 a team of which Webb was part undertook preliminary work on the Mission Control Center computer system and programs. Among other tests, an orbit computation program that had been previously been used during the

[39] Turner's posting is dated 28 Feb 1995 23:31:39 GMT. Pop's originary posting is dated Mon, 19 Sep 1994 at 10:56:50 GMT. See http://www.rchrd.com/Misc-Texts/Famous_Fortran_Errors (last accessed on 23 May 2009 at 12.01 GMT).

[40] Dave Curry's posting gives a detailed account of the research originally started by John Shore on 'documented reports on "famous bugs"'. See http://www.textfiles.com/100/famous.bug (last accessed on 23 May 2009 at 12.19 GMT).

Mercury flights was checked for accuracy, and the person conducting the tests came across a statement in the form of 'DO 10 I=1.10'.[41]
Webb writes:

> This statement was interpreted by the compiler (correctly) as:
>
> DO10I = 1.10
>
> The programmer had clearly intended:
>
> DO 10 I = 1, 10
>
> After changing the `.' to a `,' the program results were corrected to the desired accuracy.
>
> (http://www.rchrd.com/Misc-Texts/Famous_Fortran_Errors, last accessed on 23 May 2009 at 12.01 GMT)

Apparently, the program's answers had been accurate enough for the sub-orbital Mercury flights, so no one suspected a bug until the programmers tried to make it more precise in anticipation of later orbital and moon flights. Thus, the error seems to have been found under harmless circumstances and was never the cause of any actual failure of a space flight.

What I want to emphasize by retelling this story is how the misguided introduction of a dot ('.') rather than a comma (',') into the source program - a human error on the part of the programmer – resulted in a perfectly 'correct' program statement which could be 'tokenized' by the compiler according to the rules of substitution of FORTRAN language. However, the dot functions as a rupture in the FORTRAN string and it leads to a very different tokenization than the one that would be obtained if the dot was replaced by a comma. With a comma, the string would be broken down by the compiler into the following tokens:

[41] Project Mercury's sub-orbital flights took place in 1961 and its orbital flights began in 1962.

DO

10

I

=

1

10

Whit a dot, the string is in turn broken down into the following tokens:

DO10I

=

1.10

Both sequences of tokens are correct according to the rules of substitution of FORTRAN. However, when executed, the first sequence of token leads to the repetition of the loop for ten times, while the second leads to an infinite loop – that is, to the repetition of the loop which goes on *for ever*.[42] Thus, ultimately, a perfectly linearized string results in the perfect execution of a sequence of actions that potentially leads the computer system to disaster. In other words, execution (or, in the technical jargon, 'run time') is the moment in which apparently successful linearization leads to uncontrollable consequences.[43] Linearization does not therefore ensure the perfect calculability of the consequences of technology (and, broadly speaking, of the future). In order to explain how this happens, let me now return to my analysis of the compiler and explore how the source program is finally re-inscribed as circuitry.

[42] Technically, the FORTRAN statement containing the comma means 'repeat the action (represented by "... CONTINUE") while increasing the value of the counter "I" from 1 to 10, then stop', while the statement containing the dot means 'assign the value 10.1 to the variable "DO10I", then go on repeating the action "... CONTINUE" forever'. However, there is no need to recur to an explanation based on meaning to clarify the different executions of the two different pieces of code. As I will show in the rest of this chapter, the tokenization leads to different re-inscriptions - that is, to different configurations of the circuitry of the system resulting in different computer behaviours.

[43] The circular time of the loop is still linear time. This makes it even clearer how linearization cannot completely rule out the unexpected.

Eventually, the intermediate code generated by the compiler is re-inscribed by the 'code generator' (the last phase of compiling) into 'the target language'. Aho et al. explain:

> The code generator takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.
>
> (Aho et al. 2007: p.10)

Importantly, this complex passage clarifies how, after all the above re-inscriptions, the source code is now inscribed ('mapping' is just another term for 're-inscription', or 'replacement') in the computer memory in a form called 'machine code'. One must be reminded at this point that *all* the previous transcriptions were equally inscribed in the computer memory – for instance, in the initial string of the source code, a blank was encoded as the ASCII code for blank. But all these previous inscriptions were so complex that it would be impossible to follow them in this chapter. Now that we have reached a transcription in 'machine code', though, it is much easier to show how this string is inscribed and what happens during the subsequent stages of re-inscription (the so-called 'execution', or 'run-time') – in other words, to find out what kind of re-inscription execution is. However, the above passage also contains a number of cryptic terms such as 'registers', 'memory locations' and 'variables' that need to be explained before I can start discussing the re-inscription of the program in machine code. Such an explanation, in turn, requires an understanding of the way in which what is usually called 'hardware' works. Let me then stray a little from the analysis of compilers in order to clarify this point.

In Chapter Three I pointed out that the distinction between hardware and software is a shifting and unstable one. I also showed how such a separation – just like the separation between different phases of software development – was also motivated by a division of labour. In the late 1960s the distinction between hardware and

software, as well as the conceptualization of hardware according to the so-called Von Neumann model, was almost universally accepted – and is still in use today.[44] A computer is generally understood as composed of many different interconnected parts – namely, the input/output peripherals, the primary and secondary memory, and the Central Processing Unit (CPU) (Burrell 2004: 5). The peripherals are devices such as keyboards, screens and printers connected to and controlled by the computer but external to the CPU that allow for the transfer of information between the computer and the outside world. The CPU and the memory enable the storage and elaboration of data and programs. These parts are of particular interest for my argument here. The CPU (or processor) basically consists of processing and control circuitry, and it is conventionally considered the computer's 'brain'.[45] Functionally, the CPU is divided into a control unit, which obtains instructions and send signals to execute them, and the Arithmetic-Logic Unit (ALU), which executes mathematical computations and logic functions.

However, according to Von Neumann, without memory the CPU would be useless, since it only 'executes' (and I will explain in a moment what 'execution' means) the operations described by a program, such as the one we have seen above, stored in the memory. The computer's primary memory is made up of a so-called RAM (Random Access Memory) and a ROM (Read Only Memory). The latter is programmed by the computer manufacturer and cannot be modified by computer users, including programmers, while the former is the primary working memory of the computer, where programs and data are stored in a way that makes them easily accessible by the CPU. The RAM is a volatile memory: when the computer is switched off, all the data and programs residing in the RAM are lost. For this reason, there must exist a few permanent programs in the ROM that are executed when the computer is switched on again, in order to start up the whole system ('bootstrap programs').[46] The RAM and the ROM are both structured in 'cells', or

[44] The Hungarian-born mathematician John Von Neumann produced his key report on the structure of computers in 1945. The most notable aspect of the 'Von Neumann computer' regards memory. In fact, Von Neumann's was the first computer that could store programs and data in the same memory.

[45] A processor constructed entirely as a very large electrical circuit (called an *integrated circuit*) on one single chip of silicon (colloquially called a *chip*) is called a *microprocessor*. What we call a 'computer' these days would be more accurately named a *microprocessor-based computer system*, or micro-computer (Burrell 2004: 5).

[46] Curiously, the term 'bootstrap' refers to the famous literary episode in which the Baron of Munchausen lifted himself in the air by pulling at his own boots' straps.

'memory locations', univocally identified by an address - namely, a progressive number usually expressed in binary or hexadecimal code.[47] Every memory location 'contains' a value, expressed as a sequence of bits (Burrell 2004: 110) – but again the expression 'contains' is a metaphor. In fact, a memory location is not a box: it is a small logic circuit (called a 'flip-flop') which is designed in such a way that it preserves its value (either '0' or '1', the presence or absence of electrical current) over time. The secondary memory comprises all those devices for the storage of information that are external to the primary memory (such as hard and floppy disks, CDs and DVDs). Importantly, the ALU is also provided with small internal memories called 'registers', devoted to temporarily storing values and external memory addresses. The RAM and the internal memory of the CPU – that is, different types of registers – are essential to the understanding of code execution. Furthermore, all these hardware components are connected to one another through internal electrical pathways, called 'buses', along which binary signals are sent. In other words, a bus either transports ones (passage of current, also codified as TRUE) or zeroes (absence of current, also codified as FALSE). There are fundamentally three types of bus: the 'address bus' (which uni-directionally sends addresses from the CPU to the memory or to the peripherals), the 'data bus' (which sends data back and forth between the CPU and the memory or the peripherals) and the 'control bus' (which carries the control units' signals). These three types of buses, the CPU registers and the logical circuitry of the ALU are all that is needed to explain whether and in what way the execution of code can be interpreted as a process of re-inscription.

Now that I have illustrated the indispensable terms describing the computer circuitry, let me return to the example of a source program written in Java language, in order to look at its re-inscription in machine code. It is worth noting at this point that, after having been broken into lexical units, transformed into tokens and re-inscribed as intermediate code, the statement 'x=x+y;' that constitutes the central part of the program

[47] I will explain the difference between these different ways of expressing numbers later on in the chapter.

```
int x; int y; int z;

{
x=x+y;
}
```

has finally been re-inscribed in the computer's secondary memory (for instance on the hard disk) as the following:

```
1       MOV AX,[202]
2       ADD AX,[204]
3       MOV [200],AX
4       .....
.................
200     10
202     3
204     4
```

As we have seen, the above language is called 'assembler'. Unlike binary code, assembler contains 'mnemonics', or strings (abbreviations or words) that make it easier for a human reader to remember a complex instruction.[48] We shall see in a moment how, for instance, the mnemonic MOV stands for a complex process that ultimately 'moves' data to a memory location. Every line of the program above is called an 'instruction'. It is worth noting here that the brief FORTRAN statement 'x=x+y;' has become a long sequence of assembler instructions. However, and once again it is important to remember that the above list of six assembler instruction is just one possible re-inscription of 'x=x+y;'. Another possible re-inscription is the one that I offer in the following paragraphs.

In order to be executed, the program, which is stored in the secondary memory of the computer (e.g. in the hard disk), is 'loaded' (that is, re-inscribed) from the hard

[48] This is in fact how the assembler language was created: it was a way to substitute (or re-inscribe) complex strings of binary digits with the more human-friendly strings called mnemonics.

disk into the primary memory (RAM), starting from a certain memory address. This initial address is also copied into a register of the CPU called Program Counter (PC), which keeps track of the execution process by storing the address of the next instruction to be executed. The instruction currently being executed is re-inscribed into another register of the CPU, the Instruction Register (IR). The program is executed one instruction at a time, from the beginning to the end. I want to emphasize here how both the assembly program and the process of execution are still dominated by the same characteristics of *linearity* and *discreteness* that I have already pointed out in my analysis of formal grammars and programming languages. The assembler program is still sequenced in time. Moreover, for every single instruction, a number of re-inscriptions are carried out in a sequence which is called the 'Fetch-Execute Cycle' (Burrell 2004: 135). A technical manual would describe these actions as follows: the processor *fetches* (or 'loads') the first instruction from memory into the IR, it recognizes it as either an arithmetical or logical operation, or as a 'move' instruction, and it executes it by activating the apposite circuit and then by inscribing the result in another memory location. What I want to highlight here is that the Fetch-Execute Cycle consists of successive re-inscriptions of bits and strings of bits *in/as* the computer circuitry. Even more importantly, the description of the Fetch-Execute Cycle that I have just provided takes the form of a narrative. Such a narrative is itself one of the possible re-inscriptions of the FORTRAN instruction:

```
x=x+y;
```

and of the assembler instructions:

```
1       MOV AX,[202]
2       ADD AX,[204]
3       MOV [200],AX
4       .....
.................
200     10
202     3
204     4
```

Let me now examine this narrative in more detail, in order to help us understand in what way the linearized sequence of assembler instructions leads to changes in the relevant computer circuitry and in what way these changes are *ordered in time.*

In the above assembler code, a memory location in the RAM is associated with each of the variables *x, y* and *z*; while each memory location contains a bit string. We shall see in a moment that these bit strings are differently sequenced (technically, 'coded'), depending on the logical value of the variables. For the sake of clarity, let me also presuppose that *x*, *y* and *z* respectively have the numerical values of '10', '3' and '4' before the execution of the program. After the execution, *y* and *z* will remain unchanged, while *x* will have the value of '7' (namely, the sum of 3 + 4). Let me also presuppose that the variables will be stored in the memory addresses 200, 202, 204. In the example provided above, AX is the name of a register of the CPU.

When the Fetch-Execute Cycle begins, the Program Counter contains the address of the first instruction to be executed ('1'). This value ('1') is copied (in other words, re-inscribed as a sequence of binary impulses) into the address bus and transported to the primary memory. In turn, the instruction 'MOVE AX,[202]', stored (inscribed) as a string of bits in the memory location identified by '1' is transported back to the CPU – which means that this instruction is re-inscribed as a sequence of binary impulses (again, the binary *inscription* of the instruction 'MOVE AX,[202]') that travel on the physical pathway of the bus towards the CPU. Here the circuits that constitute the IR change accordingly, *becoming* the binary code of the instruction 'MOVE AX,[202]' itself. This is why I have stated above that bits and strings are inscribed *in/as* computer circuitry. The fetch phase ends with the increment of the PC, which now has the value '2' - that is, the value of the next memory location.

The binary code for 'MOVE AX,[202]' is now inscribed in/as the IR, and it describes the action 'copy the value contained in the memory location 202 (which corresponds to the value of variable *y*) into the register AX'. To execute this instruction, the address 202 (coded as a string of bits) is transported on the address

bus; subsequently, the content of the corresponding memory location (the binary encoding for '3') is retrieved and transported onto the data bus to the CPU and re-inscribed in the register AX. Now '3' is inscribed in binary form in/as the register AX. Thus the execution (and the Fetch-Execute Cycle) of the first instruction terminates.

Next, the Fetch-Execute Cycle of the second instruction begins. The value contained in the PC ('2') is transported from the CPU to the primary memory on the address bus. The instruction 'ADD AX,[204]' is carried to the CPU on the data bus and re-inscribed into the IR. The PC is incremented to '3', and the *fetch* phase terminates. The instruction 'ADD AX,[204]' - 'add the value in the registry AX to the value in the memory location 204, and store the result in the registry AX' - is executed. This means that this time the address 204 travels from CPU to memory, through the address bus, and that the content of the memory location identified by 204 (the value '4') is transported back from memory to CPU, through the data bus. The arithmetical circuitry of the ALU calculates the 'ADD' operation on the two values '3' and '4' and the result ('7') is stored in the AX register. This means that now the circuitry constituting the AX register, which encoded the value '3', has its polarities changed to 'become' the binary encoding (inscription) of the value '7'. The execution of the second instruction is thus completed and the *fetch* phase of the third one begins.

Again, the value of the PC ('3') is transported to the memory on the address bus; the corresponding instruction 'MOV [200],AX' is transported from memory to CPU and copied in the IR, and the PC is incremented to '4'. The 'MOV [200],AX' ('copy the value of the registry AX into the memory location identified by '200') is executed. This time it is necessary to transport the value '200' on the address bus and the value '7' stored in AX on the data bus *at the same time.* It becomes clear at this point that all the above activities need to be coordinated in time – and for this reason the CPU contains a 'clock', or an electronic circuit that generates evenly spaced impulses *to keep track of time.* Ultimately, then, this is the internal computer time of program execution. Eventually, however, values '7' and '200' reach the central memory, and the binary code for '7' is stored in the memory location identified by '200'.

Importantly, the above re-inscription of the piece of code

```
1       MOV AX,[202]
2       ADD AX,[204]
3       MOV [200],AX
4       .....
.....................
200     10
202     3
204     4
```

takes the form of a detailed narrative precisely because it needs to give an account of the sequence of changes taking place within the computer circuitry – a sequence ordered *in time* and regulated by the computer internal clock (i.e., a mechanism that keeps track of time through generating physical 'traces' in the form of evenly spaced impulses). Once again, though, such a 'narrative' is just another possible re-inscription of the assembly code and of the FORTRAN statement 'x=x+y;'.[49] An alternative to such a narrative would be to 'observe' the physical changes of the relevant computer circuitry – although this is the point where software becomes 'unobservable'. As I pointed out in the Introduction, this does not amount to saying that the above narrative is the 'representation' of something unperceivable that nevertheless 'really' happens in the computer, or that such a narrative is – in Alexander Galloway's terms – an extended 'metaphor' for the 'real', 'physical' level of circuitry. On the contrary, what I want to emphasize here is that the intimate engagement with software that I have argued for throughout this dissertation must be freed of any over-simplified implication of directness. The working of mediation in our engagement with software must always be kept in mind. And yet, the above narrative *is* one possible re-inscriptions of the changes in computer circuitry that can also be re-inscribed as ''x=x+y;' or as

[49] Although with the term 'narrative' I refer to my description of the Fetch-Execute Cycle, the assembly code could also be thought of as a piece of narrative. My description of the Fetch-Execute Cycle – that is, the story I tell *about* code – re-inscribes in a natural language (English) the sequencing of time that in the assembly code (or even in the FORTRAN code) is inscribed as a string.

```
1       MOV AX,[202]
2       ADD AX,[204]
3       MOV [200],AX
4       .....
.................
200     10
202     3
204     4
```

Furthermore, the narrative I have provided makes apparent how the interruptions in the string of the source program are re-inscribed as changes of voltage ordered *in time*. Nevertheless, it would not be correct to assume that only such a narrative (or the 'actual' execution of code in/as computer circuitry) can give an account of time and of the way in which the 'blanks' lead to the tokenization of code and, ultimately, to its execution at 'run-time'. Rather, as I have shown earlier on in this chapter, temporality is always present in the re-inscriptions of code – for instance, in the orientated arrow of the transformation rules of formal grammars, or in the space of the page which presupposes the orientated movement of the eyes of the reader. I want to signal here that Lev Manovich's idea that 'software studies' should not 'read code' because software becomes 'culturally visible' only when executed makes the rather over-simplified assumption that code is a static object which only becomes dynamic (and therefore interesting from the point of view of software studies and broadly speaking of media and cultural studies) when executed (Manovich 2008). In fact, this chapter aims to provide an argument to the contrary. To understand this point better, let me examine another possible re-inscription of the above assembler code.

The narrative I have offered earlier on in this chapter is in itself quite simplified, because it does not take into account that everything in the computer is ultimately inscribed as strings of bits - that is, as polarized (on/off, '0'/'1') circuitry. Moreover, while memory is conventionally subdivided into bytes (standard strings of eight bits), what is usually considered as the minimum memory unit is not the byte but the 'word'. Here the meaning of 'word' is different from the one deployed by Hopcroft and Ullman above, and it is conventionally considered as expressing a

hardware characteristic. In fact, the word has the same capacity (or length) of the data bus and of the CPU registers. It usually coincides with the number of bytes used to memorize an integer number. Therefore, the values of the integers we have seen above ('3', '4' and '7') are inscribed in the computer as strings of bits. Similarly, the instructions of the above examples are also inscribed the computer as strings of bits. The length of the string depends on the processor. For instance (in an INTEL 8086 processor) the instructions of the above example are encoded respectively on the following numbers of bytes:

MOV AX,[202]	3 bytes
ADD AX,[204]	4 bytes
MOV [200],AX	3 bytes

Thus, these assembler instructions can also be re-inscribed in one of the following ways:

Memory addresses	Machine language	Assembler language
0CA0:0100	A10202	MOV AX,[202]
0CA0:0103	03060402	ADD AX,[204]
0CA0:0107	A30002	MOV [200],AX

The above table simply means that the content of the memory address '0CA0:0100', which can be inscribed as 'MOV AX,[202]', can also be inscribed as 'A10202'. But again one must keep in mind that these are only *some* of the possible re-inscriptions of the string 'MOV AX,[202]'. In fact, both '0CA0:0100' and 'A10202' appear here in the so-called hexadecimal form. The hexadecimal form is nothing but an *alphabet* composed of 16 different digits (the numbers from '0' to '9' plus the letters 'a', 'b', 'c', 'd', 'e', 'f') that can *re-inscribe* the decimal numbers according to the following table:

Hexadecimal notation	Decimal notation
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15
10	16
11	17
.....	.....

Simple algebrical operations allow the translation (re-inscription) of the hexadecimal notation into the binary one:

Hexadecimal notation	Binary notation
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

The above table shows how the symbol '1' of the hexadecimal alphabet can be re-inscribed as the string '0001' of the binary alphabet, and so on. Therefore, the program of our example can be re-inscribed as the following:

Memory address	Machine language	Assembler language
0000110010100000:0000000100000000	10100001000000100000010	MOV AX,[202]
0000110010100000:0000000100000011	00000011000001100000010000000010	ADD AX,[204]
0000110010100000:0000000100000111	10100011000000000000010	MOV [200],AX

To understand this table one must be reminded that one byte can encode two hexadecimal digits – that is, that the binary string '0001' is actually re-inscribed in the computer memory as '00000001'. What appears obvious from the above table is that the binary representation of our program – albeit very close to the functioning of software *as* circuitry - is quite long and difficult to handle. Nevertheless, this was

the situation before mnemonics were introduced, the assembler language was developed and high-level languages were designed. I have therefore come full circle to the concept of programming language, formal grammar and compiler that I have examined in Hopcroft and Ullman's textbook. After all, as we have seen at the beginning of this chapter, less than fifty years ago no high-level programming language was available. In order to be able to develop FORTRAN, Backus first needed to develop a formal grammar, or a method that allowed him to describe (or rather, *inscribe*) language in some form, so that it could be externalized on paper in a way that could also be used as a foundation for the development of a compiler. In other words, he needed a form that could be *re-inscribed* many times in order to become software, which would then be capable of compiling a FORTRAN program.

However, it is hopefully quite clear now how the narrative that I have offered earlier on in this chapter while describing the Fetch-Execute Cycle constitutes in itself a simplified but meaningful re-inscription of code. In fact, such a narrative makes visible in what way the linearized string of code - which undergoes many re-inscriptions during the phases of compilation and execution, which in turn are made possible by its interruptions or 'blanks' - becomes a sequence of changes in computer circuitry. It also makes visible how the ruptures in code work, or, how the *iterability* of code – that is, its very repeatability – *constitutes the time* of code execution. One must be reminded at this point that such a narrative also makes clear how writing about code is always self-reflexive. In other words, writing *about* code is always also *a re-inscription* of code. In this sense, this dissertation can be seen as much a part of software technical manuals, specifications, FORTRAN programs and binary code. Even more importantly, my investigation of code execution shows how a perfectly linearized piece of code can nevertheless give rise to unexpected consequences. For instance, as we have seen in the example above, it can lead to the infinite repetition of a certain sequence of actions or changes in computer circuitry. In other words, since it unavoidably implies iterability, the process of linearization can never exclude the possibility of the unexpected – or of generating consequences that escape predictability. When such unexpected consequences occur, as I pointed out in Chapter Four, it becomes necessary to make a decision. It has to be decided whether such unexpected consequences are acceptable or not, whether they should

be considered malfunctions that need to be fixed or possibilities that can be incorporated into the system – or even perhaps risky opportunities that can lead to radical changes in the system itself.[50]

To sum up, in this chapter so far I have discussed how software is constituted in the theory of programming languages of the late 1960s as a process of unstable linearization. In this theory, software is understood through concepts derived from linguistics – namely language, grammars and the alphabet. I have paid such a close attention to this conceptualization not in order to argue that software is a form of language or of writing – at least, not 'writing' in any historical sense of the word (or what Derrida would call 'writing in the narrow sense'). On the contrary, my argument throughout this chapter, as well as the thesis as a whole, has been that, in historically specific circumstances, 'software' is constituted in relation with (historically specific) concepts of language and writing. For instance, in the theory of programming languages it is defined by using the terms of Chomskyian linguistics, while in Software Engineering the three terms 'software', 'writing' and 'code' are constituted together within a methodology for the control of the time of industrial software production.

Even more importantly, I would like to alert the reader to the fact that neither in this chapter nor throughout this dissertation have I made a knowledge claim with regard to what software *is*. I have rather asked what software *does* – and, with a view to that, I have investigated historically specific (or rather, singular) instances of software in order to show how software works. My investigation has brought to the fore the fact that the conceptualization of software is always problematic, since in every singular understanding of software there are points of opacity. In other words, parts of the conceptual system which structures software have to remain unthought in order for software to exist. The most important point of opacity that has emerged from my analysis is the conceptualization of software in terms of instrumentality. A sustained attempt to define software in these terms can be found in Software Engineering in particular as well as in the theory of programming languages. Such a

[50] As I argued in Chapter Four, the emergence of open source in the 1990s was an unexpected consequence of the conception of programming of the 1970s and 1980s that led to a radical change in software design.

definition – i.e., the understanding of software as a tool – presupposes that software is controllable, that its development and uses can be planned and that the risks and consequences implicit in software can be foreseen. This concept of software is based on the general understanding of technology in the Western philosophical tradition: the Aristotelian idea that technology is a tool that must be mastered by humans to pursue certain ends. Consistently with this Aristotelian line of thought, in Software Engineering and in the theory of programming languages software is defined in binary terms (through oppositions between conceptual/material, 'what'/'how', specification/implementation). Also consistently with this tradition, software is understood in linear terms, both in the linearization of programming languages and in the linearization of the time of software development. This philosophico-technical conjuncture is what, in the words of Timothy Clark, Derrida understands as the 'complicity of technology with metaphysics' (Clark 2000: 248).

However, as I have attempted to show here, a critical analysis of software shows that software cannot be fully conceptualized within the Aristotelian framework. In fact, in the theorizations of software that I have examined, software escapes its definition in Aristotelian terms and appears to function rather as a material inscription capable of generating unforeseen consequences. This functioning of software emerges through points of opacity – for instance, the undoing of the distinction between hardware and software, or the unavoidable return of iteration in the process of software development (a process that can never be completely linearized), or in the impossibility of deciding whether an unexpected anomaly in software is a malfunction or a welcome variation.

Even more importantly, it must be kept in mind that these observations are not automatically valid for *all* software. Every instance of software needs to be studied in its singularity, and problematized accordingly. What is more, the opacity of software cannot be dispelled merely through an analysis of what software 'really is' – for instance by saying that software is 'really' just hardware (Kittler 1995), or by unmasking the economical interests behind it, or by contrasting the theory of software against the personal self-accounts of programmers (Fuller 2003). Rather, one must acknowledge that software is *always* both conceptualized according to a metaphysical framework *and* capable of escaping it – and the singular points of

opacity of singular instances of software need to be brought to light. (Or, as Derrida would have it, the process of deconstruction needs to be carried out.)

This process of the problematization of technology is creative, productive and politically meaningful. In fact, it shows that, since not everything in technology can be thought (or fully conceptualized within one consistent framework), and since there always remain points of opacity, technology also always brings about unexpected consequences. This is the problem that I have started from, or the question that, following Stielger, I have emphasized in the Introduction – namely, that we need to make decisions about a technology which is always somehow opaque. These decisions are ethical and political and they influence our very existence as human beings – not just as users of tools and machines but also as beings that co-emerge and co-evolve with technology. If one takes into account the unavoidable opacity of technology, no Habermasian way out of this dilemma can be imagined – namely, it is not enough for policy makers and citizens to make 'informed' decisions regarding technology. Of course, such decisions are inevitable and necessary, but it must also be kept in mind that not everything in technology is calculable, and that therefore every decision about technology is an assumption of responsibility for something that we cannot actually foresee. And yet a decision must be made, and responsibility needs to be taken. The more ethical decisions are the ones that take into account – or, in other words, do not mask - this dilemma and that give account of their own reasons. By making these decisions *responsibly*, new possibilities are being opened, while others are being foreclosed. In fact, such decisions do not just affect technology; they also change our experience of time, our modes of thought and, ultimately, our understanding of what it means to be human. In this sense, we gain a sense of who we are only through technology. For this reason, I want to suggest that paying continuous and careful attention to the singularity of new technologies is the only way of guaranteeing the development of a politically informed and responsible agenda for the study of digital technologies.

Conclusions

The Unforeseen Consequences of Technology

'Fighting an alien robot? That was me! And it was amazing!', boasts Susan Murphy soon after having defeated an alien robot probe in San Francisco with the help of a gelatinous blue blob and a gay fish-ape hybrid. She gleefully proceeds to enumerate all the different ways in which being a monster is an extremely appealing and rewarding status for an American girl of her age. All the while, the group of freaks that surround her – which includes a mad scientist and a perambulating insect chrysalis – marvel at the discovery of their own virtues and talent.

I would like to propose a brief reading of the computer-animated 3D feature film from DreamWork Animation and Paramount Pictures, *Monsters vs. Aliens*, as a kind of 'alternative summary' of my dissertation here. *Monsters vs. Aliens* was released in March 2009. In this film Susan Murphy, a young woman from Modesto, California, is hit by a radioactive meteor on the day of her wedding, thus absorbing a rare substance called quantonium that mutates her into a giantess. Immediately captured by the US military and classified as a 'monster', she is imprisoned in a top-secret facility directed by General W.R. Monger where other 'monsters' are kept in custody, among whom are B.O.B. (Bicarbonate Ostylezene Benzonate, an indestructible gelatinous blue blob without a brain), Dr. Cockroach, Ph.D. (a mad scientist with a giant cockroach's head), the Missing Link (a 20,000-year-old amphibious fish-ape hybrid) and Insectosaurus (a 350-foot grub). When an alien named Gallaxhar attacks the Earth with his gigantic robotic probes and an army of clones of himself, General Monger persuades the president of the United States to

deploy the monsters as military weapons. Having accepted the mission with the promise of freedom if they succeed, the monsters manage to destroy the alien robotic probe that Gallaxhar has sent to San Francisco. During the fight Susan discovers that she possesses an unexpected strength and that she is also invulnerable to Gallaxhar's weapons. Having been freed, Susan happily returns to Modesto - only to be rejected by her fiancée (who claims that he cannot be married to a woman who overshadows him) while unwittingly her monstrous friends disseminate panic in the neighbourhood. Initially sad and dispirited, Susan suddenly realizes that becoming a monster has actually enriched her life, and she fully embraces her new 'amazing' lifestyle and her newly formed bond with the other monsters. After a final epic fight Susan and her gang completely defeat Gallaxhar and his cloned army, and are eventually acclaimed as heroes. In the last scene of the film, they are alerted to the fact that in the surroundings of Paris a snail has fallen into a nuclear power-plant and is growing into a giant due to nuclear irradiation. They then fly off on a mission to protect the Earth from the new enemy.

In the context of the overall argument of my dissertation, what is particularly interesting about *Monsters vs Aliens* is that in the movie the monsters function first and foremost as a figure of the unexpected consequences of technology.[1] In fact, they all *come into existence* as unforeseen effects of technology: they are the

[1] The queer theorist Judith Halberstam has been recently constructing a 'queer' archive of 3D animated features (Halberstam 1997), where the term 'queer' means that such features incorporate a politically subversive narrative cleverly disguised in a popular media form aimed at children. For instance, according to Halberstam the CGI animated film of 2003, *Finding Nemo*, depicts the title character - a motherless fish with a disabled fin – as a 'disabled hero' and links the struggle of the rejected individual to larger struggles of the dispossessed (Nemo leads a fish rebellion against the fishermen). Halberstam proposes the term 'Pixarvolt' to indicate movies depending upon Pixar technologies of animation and foregrounding the themes of revolution and transformation. For her, the Pixarvolt films use the individual character as a gateway to stories 'of collective action, anti-capitalist critique, group bonding and alternative imaginings of community, space, embodiment and responsibility' (Halberstam 2007: non-pag.). In a sense, it could be said that the monsters in *Monsters vs Aliens* yield themselves to a queer reading - actually, queer references seem to have become quite commonplace in animated features. For instance the Missing Link is a parody of excessive masculinity (notwithstanding his machismo and his gung-ho attitude to fight, he is comically out of shape) and has a gay bond with Insectosaurus; the monsters perform part of their first battle against Gallaxhar on a stolen San Francisco bus directed to the Castro, and, even more tellingly, the transformation of Susan into a monster frees her from all heterosexual social expectations and places her in a queer alliance within other social outcasts. Nevertheless, it is debatable whether the narrative of the film can be read as subversive, since the monsters' community seems not so much to constitute an alternative to the mainstream society as a weapon in the hands of the American government – although one could argue that such subversive narratives are at their most intriguing when they are apparently neutralized. As I will show in a moment, the neutralization of the monsters in *Monsters vs Aliens* is in fact apparent, but to understand this point the monsters must be viewed in their relationship with technology.

unpredictable outcomes of experiments gone wrong. B.O.B. was mistakenly created by injecting a genetically-modified tomato with a chemically-altered ranch dressing; Dr. Cockroach ended up with an insect head and the ability to climb walls while subjecting himself to an experiment in order to gain the longevity of a cockroach and the mad scientist is the figure of the experiment gone wrong *par excellence.* Insectosaurus, originally a one-inch grub, was transformed into a giant after being accidentally invested by nuclear radiation. Even the Missing Link could not have been found frozen in a lagoon and thawed out by scientists without some help from technology.[2]

However, in *Monsters vs Aliens* the monsters are also 'domesticated' – or rather, they are kept under custody by the American government and later on transformed into weapons. In other words, the film seems to imply that technology needs to be controlled in order to be made useful (i.e., it has to become a tool). But in order to be successfully deployed as weapons monsters must be released from custody – that is, in order to be 'used', technology must be set free. In turn, once set free, technology escapes instrumentality. In fact, as we have seen from the synopsis above, it is by fighting Gallaxhar that Susan discovers her unexpected physical strength. Similarly, during the final battle against the aliens Insectosaurus apparently dies, while actually undergoing a metamorphosis from a chrysalis into a beautiful butterfly. Ultimately, though, the monsters are still kept under control: they constitute an American military team, albeit a very special one. It is here that aliens find their place in the film narrative: a relationship which would otherwise be quite uncomplicated (humans detain and domesticate dangerous monsters) finds its third term in the aggressive threat from the outside. Aliens provide an enemy and help construct the narrative of the American fight for democracy against (alien) totalitarian regimes. Even though it occasionally makes fun of the American

[2] Ostensibly the film here taps into the popular tradition of superheroes - that is, fictional characters endowed with superhuman powers and devoted to fighting crime or injustice that have dominated American comic books for decades and have subsequently crossed over into other media. The so-called 'origin stories' associated with superheroes, which explain the circumstances by which the characters acquired their exceptional abilities, often involve experiments gone wrong. For instance, Spider Man (Peter Parker) got bitten by a radioactive spider during a science demonstration at school when he was a teenager, while the Fantastic Four (Reed Richards, Sue and Johnny Storm and Ben Grimm) were accidentally exposed to cosmic rays during a space mission (Reynolds 1994). In other words, it can be said that superheroes themselves embody unexpected consequences of technology which have nevertheless been reframed into a narrative of fight against evil.

government (General W.R. Monger's name is a pun on the word 'warmonger', while the inept president of the United States is always on the verge of launching a nuclear attack by pressing the wrong button), the film still embraces a narrative that legitimates the Unites States as the world super-power.

The point of opacity of the film is to be found at its end: in the final scene the monsters set off to Paris to fight the gigantic snail which has broken into a nuclear plant – but should the snail be understood as an alien or a monster? Since it is presented as a threat against which the monsters are supposed to fight, it must be an alien. And yet, since clearly it is an unexpected effect of technology (actually, accidental nuclear irradiation is one of the most common origin stories of superheroes and is very similar to Insectosaurus' story) the snail must be a monster and in principle it should not be fought but rather helped out or maybe even recruited as part of the team. With a revealing lapse, a Wikipedia entry (http://en.wikipedia.org/wiki/Monsters_vs_Aliens) recounts how at the end of the film 'the monsters are alerted to a *monster attack* near Paris and fly off to combat the new menace' (italics mine). In Derridean terms, it could be said that the snail is the incest taboo of *Monsters vs Alien*.[3] In other words, the snail is the point where the instrumentality of technology undoes itself, because technology is always both a monster and an alien, an instrument and a threat, a risk and a promise. The unexpected is always implicit in technology, and the potential of technology for generating the unexpected needs to be unleashed in order for technology to function *as* technology. The attempt to control the unexpected consequences of technology is ultimately destined to fail - and yet it must be pursued for technology to exist. For this reason, every choice we make with regard to technology always implies an assumption of responsibility for the unforeseeable. As demonstrated in my critical reading of the history of software in the discipline of Software Engineering throughout the chapters of this thesis, we can see that whenever one makes decisions about technology, one has to remember that technology can always generate consequences that escape predictability. Thus, to think of technology in a political

[3] In *Of Grammatology* (1976) Derrida famously shows how the incest taboo is the unthought of structural anthropology – that is, a concept that cannot be thought within the conceptual system of the discipline because it escapes its basic opposition between nature and culture. In fact, the incest taboo appears to be neither completely natural nor totally cultural, thus constituting the 'point of opacity' of structural anthropology.

sense we must first and foremost remember that technology cannot be thought from within the conceptual framework of calculability and instrumentality.

To present a more conventional summary of the work undertaken in this dissertation, I have attempted here to propose an analytical framework for the cultural understanding of the group of technologies commonly referred to as 'new' or 'digital'. I have argued that a 'demystification' of new technologies – or the dispelling of the 'opacity' that still surrounds them and that constitutes one of the main obstacles in their conceptualisation - is crucial if we are to engage with them on both the cultural and the political level. I have also argued that an instrumental understanding of technology is not sufficient to make sense of new technologies, and especially of software-based technologies.

In Chapter One, drawing on the work of Bernard Stiegler (1998), I questioned the dominant philosophical conception of technology based on the Aristotelian thought, which substantially reduces technology to a mere instrument.[4] Still following Stiegler's argument, I turned to the work of those thinkers who have distanced themselves from such an instrumental understanding and have instead proposed a view of technology as a fundamental characteristic of human beings (such as Stiegler himself, as well as Martin Heidegger and Jacques Derrida). Timothy Clark (2000) calls this the tradition of 'originary technicity' – a term he borrows from Richard Beardsworth (1996). I then argued for a radical rethinking of the conceptual framework of instrumentality if an understanding of technology is to be made possible. A pivotal role is played here by the concept of linearization, which belongs to the tradition of originary technicity and was developed by the French paleonthologist André Leroi-Gourhan (1993) and subsequently re-read by Derrida in *Of Grammatology* (1976) in the context of his own reflections on writing.

Derrida explains how Leroi-Gourhan has shown in *Le geste et la parole* that the historical perspective that associates humanity with the emergence of writing (and therefore excludes peoples 'without writing' from history) is profoundly ethnocentric. In fact, it shortsightedly denies the characteristic of humanity to

[4] *Nicomachean Ethics* 6, 3-4 (Aristotle 1984).

peoples that do not actually lack 'writing' as such, but only 'a certain type of writing' (Derrida 1997: 83), that is, alphabetic writing. What Leroi-Gourhan calls 'linearization' is precisely the emergence of alphabetic writing (Leroi-Gourhan 1998: 190-216). In his analysis of the emergence of graphism, Leroi-Gourhan highlights what he considers to be the underestimated link between figurative art and writing, and proposes the name 'picto-ideography' for such a general figurative mindframe. The term 'picto-ideography' signals an originary independence of graphism from the mental attitude that constitutes the basis of linearization.[5] In fact, graphism is not dependent on spoken language, and the emergence of alphabetic writing is associated with the technoeconomic development of the Mediterranean and European group of civilizations. At a certain point in time during this process writing became subordinated to spoken language. By becoming a means for the phonetic recording of speech, writing thus became a technology – it was placed at the level of a tool, or of 'technology' in its instrumental sense. As a tool, its efficiency became proportional to what Leroi-Gourhan views as a 'constriction' of its figurative force, pursued precisely through an increasing linearization of symbols. Leroi-Gourhan calls this process 'the adoption of a regimented form of writing' that opens the way 'to the unrestrained development of a technical utilitarianism' (212).

In *Of Grammatology*, Derrida draws on Leroi-Gourhan's view of phonetic writing as 'rooted in a past of nonlinear writing', and on the concept of the 'linearization' of writing as the victory of 'the irreversible temporality of sound' (Derrida 1976: 85). Expanding on Leroi-Gourhan, Derrida relates the emergence of phonetic writing to a linear understanding of time and history. Linearization is nothing but the constitution of the 'line' as a norm, a model. Yet, the line is *only* a model, however privileged. The linear conception of writing implies a linear conception of time, that is a conception of time as homogeneous, and involved in a continuous movement, straight or circular. Derrida draws here on Heidegger's argument that this conception of time characterizes all ontology from Aristotle to Hegel, that is, all

[5] Leroi-Gourhan is very clear that such a mindset does not correspond to writing 'in its infancy' (Leroi-Gourhan 1998:195). Such an interpretation would amount to applying to the study of graphism a mentality influenced by four thousand years of alphabetic writing – something that linguists have often done, for instance, when studying pictograms.

Western thought. Therefore, and this is the main point of Derrida's thesis, 'the meditation upon writing and the deconstruction of the history of philosophy become inseparable' (86). Derrida's reading of Leroi-Gourhan thus emphasises the relationship between technology and language, and between language and time. Writing has become what it is through a process of linearization, that is, by conforming to the model of the line. This model also characterizes the idea of time in Western thought. Questioning the idea of language as linear therefore implies questioning the role of the line as a model, and thus the concept of linear time: ultimately, it implies questioning the very foundations of Western thought. For this reason, a rethinking of technology (which is related to language and writing) entails a rethinking of Western philosophy. In sum, for Derrida what is most relevant in Leroi-Gourhan's history of writing is that it problematizes our conception of the human – in Derrida's words, 'what we believe we know of the face and the hand' (84).

Drawing on the above discussion of the philosophical premises of the notion of technology, in Chapter Two I argued that the study of software as a historically specific technology is important for a radical rethinking of the relationship between technology and the human. Following Derrida's insight on 'new technologies', and his clarification of how limiting it is to conceive them merely in terms of instrumentality (Derrida 1983), I set out to investigate how, on the one hand, software can illuminate the role of technology in the constitution of the human - that is, how it can illuminate 'originary technicity' - while, on the other hand, how the 'complicity' (Derrida's term) with the instrumental understanding of technology can still be identified to be at work in software. I also argued that it is impossible to investigate 'software' in general, since what we call 'software' takes many forms in many different contexts. Thus, I focused on the constitution of the term 'software' in relation to two other terms - 'writing' and 'code' - in the theories and practice of Software Engineering at the end of the 1960s. In Chapter Three I demonstrated that Software Engineering was constituted as a discipline in the context of the industrialization of software production and that in this discipline instrumentality took the form of the regulation of the relationship between 'software', 'writing' and 'code'.

The main line of my argument in the first part of my dissertation was that Derrida's conceptualization of writing can be put to work in the interpretation of code. Since his earliest works (in particular, *Of Grammatology*), Derrida has defined writing as a material practice. As we have just seen, for Derrida the subordination of writing to speech has meaning only within the system of Western metaphysics, the premises of which have been inherited by human sciences - particularly by linguistics. In *Of Grammatology*, Derrida focuses on the deconstruction of linguistics and of its central concept, the concept of the sign. For him, the sign is exemplary of the metaphysical devaluation of materiality, because the very possibility of the sign is predicated on an opposition between that which is conveyed (the signified) and the conveyor (the signifier) (Beardsworth, 1996: 7). The signifier is a material entity, such as a sound or a graphic sign; the signified belongs to the realm of concepts. For Derrida this opposition is the foundation of all the other oppositions that characterize Western metaphysics (infinite/finite, soul/body, nature/law, universal/particular, etc.). Deconstructing the sign is thus a fundamental and exemplary move precisely because metaphysics is derived from the domination of a particular relation between the ideal and the material. The sign constitutes the foundation of the distinction between signifier and thing, a distinction which in turn is the basis of *episteme* and therefore of metaphysics.[6]

For Derrida we therefore need to have an understanding of writing in order to grasp the meaning of orality - not because writing historically existed *before* language, but because we must have a sense of the permanence of a linguistic mark in order to recognise it. Ultimately, a sense of writing is necessary for signification to take place. However, we can have a sense of the permanence of the mark only if we have a sense of its inscription, or of its being embodied in a material surface.[7] This is what Derrida means when he says that 'the transcendental' is always impure, always already contaminated by 'the empirical'. In other words, language itself is material for Derrida; it needs materiality (or rather, it needs the possibility of an 'inscription') to function *as* language. Thus, textuality and materiality are not

[6] In Richard Beardsworth's words, 'metaphysics constitutes its oppositions (here: the non-worldly/worldly and the ideal/material) by expelling into one term of the opposition the very possibility of the condition of such oppositions' (1996: 8).

[7] In other words, although we recognize the written form of a grapheme (let's say 't') only by abstracting it from all the possible empirical forms a 't' can take in writing, we need such an empirical inscription to make this recognition possible.

opposed: materiality is the condition of signification and every code is material. Or, even better, the possibility of inscription is the very condition of code's functioning. Software itself can function only through materiality, because software is (also) code, and materiality is what constitutes signs (and therefore codes). Furthermore, writing is based on the very same possibility of material inscription, so it should come as no surprise that it has been chosen as the privileged way of accessing digital technology by Software Engineering. But if every bit of code is material, and if the material structure of the mark is at work everywhere, how are we supposed to study software as a historically specific technology? In Chapter Two I argued that the historical specificity (or, in Gary Hall's terms, 'singularity') of software resides in its constitution through the continuous undoing and redoing of the boundaries between 'software' itself, 'writing' and 'code'.

Moreover, in Chapter Three I showed how the discipline of Software Engineering emerges as a strategy for the industrialization of the production of software at the end of the 1960s and how it understands software as a process of material inscription that continuously opens up and reaffirms the boundaries between 'software', 'writing' and 'code'. Software Engineering establishes itself as a discipline precisely through an attempt to control the constitutive fallibility of software-based technology. Such fallibility – that is, the unexpected consequences inherent in software - is dealt with through the organization and linearization of the time of software development. Software Engineering also understands software as the 'solution' to pre-existent 'problems' or 'needs' present in society, therefore advancing an instrumental understanding of software. However, both the linearization of time and the understanding of software as a tool are continuously undone by the unexpected consequences brought about by software – which must consequently be excluded and controlled in order for software to reach a point of stability. At the same time, such unexpected consequences remain necessary to software's development. Through the analysis of the different stages of software development as described in the Garmisch conference report (Naur and Randell 1969) I showed how the unforeseeable consequences of software are inscribed in

software itself in all its forms.[8] In particular, they are inscribed in code by means of its characteristic of 'extensibility' and 'modularity'. The combination of extensibility and modularity constitutes a way to calculate the future of an open-ended software system – but, since nobody can anticipate what an open-ended system will do, or what can be done with it, it also keeps the possibility of the unforeseeable open. The figure of the 'user' of software represents both the instability of the instrumental understanding of software and the capacity of software to escape instrumentality through the unexpected consequences it generates.

In Chapter Four I turned to the investigation of the establishment of Software Engineering in the 1970s and 1980s as a discipline for the management of time in software development. Subsequently I presented the emergence of the open source movement in the 1990s as one of the unforeseen consequences of the Software Engineering of the late 1980s. The sequencing of time that Software Engineering proposed in the 1970s and 1980s, which was justified through an Aristotelian distinction between the essential and the accidental (the ideal and the material), ultimately gave rise to an unexpected re-organization of technology according to the open source style of programming. The framework of instrumentality that, as I showed in Chapter Three, emerged from the 'software crisis' of the late 1960s - that is, from the need to control the excessive speed of software growth - was enforced on software production through the 1970s and 1980s, and it enabled the development of software during these decades *as well as* the unexpected emergence of open source. I also showed how open source still takes Software Engineering as its model (Raymond 2000), a model in which the framework of instrumentality is once again re-enacted. The aim of open source is still to obtain 'usable' software, but 'usability' - that is, a stable version of a software system - is not something that can be scheduled in time. Rather, it is something that *happens* to the system while it is constantly re-inscribed. Ultimately, in Chapters Three and Four I argued that Software Engineering is characterized by the continuous opening up and

[8] The report of the first ever conference on Software Engineering, convened by the NATO Science Committee in 1968 in Garmisch (Germany) is considered one of the foundational texts of Software Engineering (Naur and Randell 1969). The second conference on Software Engineering was convened by NATO in 1969 in Rome (Italy); although the Rome report is also considered a foundational text, its influence on the discipline has been much more limited (Buxton and Randell 1970).

reaffirmation of the instrumentality of software and of technology. In Software Engineering, the instrumentality of software is (as Derrida would have it) 'in deconstruction': it is the unstable result of the process of technological exteriorization. And yet, since the undoing and redoing of instrumentality can go unnoticed, a 'deconstructive reading' - that is, a critical and creative problematization - of software must be actively perfomed. This active problematization of software ultimately makes 'originary technicity' most visible: it clarifies the significance of software (i.e., of singular instances of software) for the relationship between technology and the human.

For this reason, in Chapter Five I strayed somewhat from the field of Software Engineering to analyse how software is constituted in the theory of programming languages of the late 1960s (on which Software Engineering ultimately relies on) as a process of unstable linearization. In this theory, software is thought by using concepts derived from structural linguistics – namely language, grammars and the alphabet. I paid close attention to this conceptualization not in order to argue that software is a form of language or of writing – at least, not of 'writing' in any historical sense of the word (what Derrida would call 'writing in the narrow sense'). On the contrary, throughout the whole of this thesis I have argued that, in historically specific circumstances, 'software' is constituted in relation with (historically specific) concepts of language and writing. For instance, in the theory of programming languages it is defined by using the terms of Chomskyian linguistics, while in Software Engineering the three terms 'software', 'writing' and 'code' are constituted together within a methodology for the control of the time of industrial software production.

Moreover, in Chapter Five I showed how in the theory of programming languages the instrumentalization of technology and the linearization of programming languages go hand in hand, and how software exceeds both. In fact, code is defined as a process of substitution (or re-inscription) of symbols: ultimately, thus, computation is viewed as iteration, or the process of generating differences in repetition. The theory of programming languages is an attempt to manage iteration through linearization – an attempt ultimately destined to fail, since iteration is always open to variation, and therefore always risky. Actually, iteration is a

constitutive characteristic of software, just as fallibility (and the capacity for generating unforeseen consequences) is constitutive of technology. For instance, a perfectly linearized – that is, formally correct – piece of code can always lead to unexpected consequences when executed. In other words, linearization does not ensure the perfect calculability of the consequences of technology (and, broadly speaking, of the future). Since the unexpected is unavoidably part of technology – and since technology is a constitutive part of our conception of time and of ourselves – ultimately the decisions that we make about technology are of the highest political significance. Every time technology brings about some unexpected consequences, it is fundamental to decide whether this is a malfunction that needs to be fixed or an acceptable variation that can be integrated into the technological system, or even an unforeseen anomaly that will radically change the technological system for ever.[9] Such a decision is primarily political rather than technical.

This is the fundamental double valence of the unexpected as both failure *and* hope. Like Derrida's *pharmakon*, technology entails poison and remedy, risk and opportunity (Derrida 1981). Once again, the inseparability of these aspects means that, every time we make decisions about technology, we are taking responsibility for uncalculable risks. There is no Habermasian way out of this irreconcilable dilemma: no 'expert' can provide society with all the necessary information to make un-risky decisions. Technology will never be calculable – and yet decisions *must* be made. The only way to make politically informed decisions about technology is *not* to obscure such uncalculability.

In the documentary *The Net* (2003), director Lutz Dammbeck shows how obscuring the incalculability of technology leads to setting up an opposition between risk and control and between 'good' and 'bad' technology, and ultimately to the authoritarian resolution of every dilemma regarding technology. Questions such as, 'Should technology be 'democratized'?', 'Should it be made available to everyone even when it is 'dangerous'?', 'Who decides what is dangerous for whom?' are then addressed by embracing either a policy of control or a deterministic, almost paranoid fear of technology, which is also possibly combined with a Luddite stance.

[9] In Chapter Four I analysed the open source movement as such a revolutionary unexpected effect of the Software Engineering of the 1980s.

The film explores the complex story of Ted Kaczynski, the infamous Unabomber. A former mathematician at Harvard, Kaczynski retreated to a cabin in the wilderness of Montana in 1971. In 1996 he was arrested by the FBI under the suspicion of being responsible for the attacks carried out between 1978 and 1995 by an unknown individual nicknamed the Unabomber against major airlines executives and scientists at elite universities. The film complicates the narrative regarding the Unabomber (the author of an anti-technology *Manifesto*, and an ultimate figure of resistance for those who oppose contemporary technology as a form of control) by situating him within the complex and contradictory web of the late twentieth century technology.

Particularly revealing is an interview with John Taylor – an ex-NASA engineer and an admirer of Norbert Wiener, the founding father of cybernetics – which shows how the idea of calculability (and the attempt to expel the unexpected from technology) was crucial for early cybernetics. Taylor recounts how ARPA (the Advanced Research Projects Agency) was set up in 1958 by the American president Eisenhower with the goal of seeking out 'promising' research projects – in Taylor's words, projects that had 'a longer term expectation associated with them'. ARPA was instituted after the launch of the Russian space probe Sputnik in 1957, which Taylor characterizes as 'a great surprise' for the United States. The American Department of Defence set up ARPA 'in the hope that we would not get surprised again like the Russian surprised us'. The ambivalence of the term 'surprise' as both risk and promise is obvious in Taylor's words: the best research projects are the ones which hold the 'promise' of 'good surprises', which will in turn prevent the enemy from surprising us in a 'bad' way. ARPA was therefore meant to 'domesticate' the potential of technology to surprise us, that is its capacity for generating the unexpected, by subjecting 'promising' projects to control. Taylor ostensibly embraces such a philosophy of control. When, during the interview, Dammbeck mentions the Unabomber, a horrified look crosses Taylor's face and, as many of his colleagues interviewed in the film do, he refuses to speak about Kaczynski, dismissing him as a terrorist and even comparing the Unabomber's *Manifesto* to Hitler's *Mein Kampf*. When Dammbeck suggests that some people such as the Unabomber might be scared by technology and asks Taylor what he is scared of, he answers 'I am scared of Al-Qaeida… I am scared of cancer. But if we

could find a cure for cancer, we wouldn't be afraid'. According to Taylor, fear is a matter of ignorance, of 'not knowing'. By possessing more knowledge – he adds with a curious phrasing - we could 'prohibit cancer'. Taylor's revealing formulation is the ultimate expression of a desire for the technological control over nature and for the complete calculability of the future.

The idea of cybernetics as the science of control takes up a new meaning here – one related to prediction, calculation, foreseeability - if one considers, as Dammbeck does, that one of the participants in the Macy Conferences (which instituted cybernetics as a discipline between 1946 and 1953), the psychologist Kurt Lewin, conceived a project for programming humans to give them an 'anti-authoritarian personality', thus obstructing the possibility of fascism forever. Oblivious to the fact that this would be the ultimate authoritarian gesture, Lewin suggested that cybernetics could control and remap people's subconscious in order to immunize people against totalitarianism and to make authoritarian systems impossible. For him, anti-authoritarianism was first and foremost a matter of calculation, as the *control of the political future* of humanity.[10] Ironically, drawing on Lewin's project, Henry A. Murray, one of the fathers of today's assessment centres, devised a series of tests which were supposed to highlight concealed psychological tendencies by penetrating consciousness with non-surgical means - basically LSD and other drugs. Such tests were carried out by the CIA in the late 1960s at Harvard on a group of talented young male students, among whom was Ted Kaczynski. Whether those experiments led Kaczynski to the fear of occult forms of mind control, and ultimately resulted in his paranoid terror of technology is a possibility that the film

[10] This is just one of the many examples of how the concept of the 'system' of cybernetics was transferred to the social and political realm quite uncritically, starting from the 1950s. In the BBC documentary *The Trap* (2007), director Adam Curtis has shown how the idea of freedom that characterizes today's neoliberal democracies is a very limited one, mainly founded on the idea of the individual as a free agent always pursuing its own self-interest. This idea is very much based on game theory and other theories developed during the Cold War period, which had a strategic importance in determining the so-called 'balance of terror' (in which basically the enemy avoids attacking for fear of being destroyed). Driven by the necessity of anticipating Soviet moves, Nash's games were based on the belief that, *in every society*, stability could be created through distrust. Nash's equations work only if individuals are presumed to be selfish and suspicious of one another. If they start to cooperate, however, then the system becomes unpredictable. In the famous Prisoner's Dilemma it is selfishness that leads to safety. This game shows that the rational move is always to betray the other. This was the logic of the Cold War (i.e., one cannot trust the other not to cheat) – but Nash turned this assumption into a *theory of society*. In his paranoid view of the lonely human being in a hostile society, the price of freedom is distrust. Curtis' fascinating documentary shows how Nash's ideas spread to fields that had nothing to do with nuclear strategy, from R.D. Laing's psychiatric thought to George Buchanan's theories of 'public choice' which assisted Margaret Thatcher's dismantling of the welfare state.

leaves open. Importantly, however, Dammbeck's film makes a suggestion that control and uncalculability, risk and opportunity, are constitutive of technology. As Dammbeck himself states, the key to Kaczynski's tragedy is the fact that he is 'part of a system from which there is no escape'. He does not understand that, even isolated in a forest cabin, one is still part of the technological system (a cabin *is* a form of technology, after all), and that there is no 'outside' of technology.

Once again I want to emphasize here that in order to make responsible decisions about technology, one must be aware that technology (as well as the conceptual system on which it is based) can only be problematized from within.[11] Such a problematization of technology needs to be a creative, productive and politically meaningful process. In fact, the problem I started from, or the question that, following Stielger, I emphasized in the Introduction – namely, that we need to make decisions about a technology which is always somehow opaque – requires precisely such an active problematization of technology. Even more importantly, technology – as well as software - needs to be studied in its singularity. In particular, the opacity of software cannot be dispelled merely through an analysis of what software 'really is' – for instance, by saying that software is 'really' just hardware (Kittler 1995), or by unmasking the economical interests behind it (Fuller 2003). Rather, one must acknowledge that software is *always* both conceptualized according to a metaphysical framework *and* capable of escaping it, that it is instrumental *and* generating unforeseen consequences, that it is both a risk and an opportunity. The singular points of opacity of singular instances of software need to be brought to light; or, as Derrida would have it, the process of deconstruction needs to be carried out.

The decisions we make about technology influence our very existence as human beings. As I pointed out in Chapter Five, even the difference between '0' and '1' can be constituted only through a process of repetition – the same process that generates consciousness and time, as well as the possibility of making every conceptual distinction. This is why I argued there that consciousness and time are constituted by the very possibility of making the distinction between '0' and '1' and

[11] This is precisely what deconstruction allows us to do – stepping out of a conceptual system by continuing to use its concepts while at the same time demonstrating their limitations (Derrida, 1980).

that perhaps it can be said that consciousness has always been digital.[12] It is in this sense that we can say that technology is constitutive of thought (including philosophical thought) and this is the reason why it can never be definitively expelled from thought. In order to think technology in a politically meaningful way it is necessary to keep in mind that technology is constitutive of our consciousness and of our temporality. By opening new possibilities and foreclosing others, our decisions about technology also affect our future. Thus, making responsible decisions about technology becomes part of the process of the reinvention of the political in our technicized and globalized world. Rethinking technology *is* a form of imagining our political future.

[12] 'We Have Always Been Digital' is the title of Joanna Zylinska's ongoing photographic project that explores digitality as the intrinsic condition of photography (http://photos.joannazylinska.net/).

Appendix A

15

2. SOFTWARE ENGINEERING AND SOCIETY

One of the major motivations for the organizing of the conference was an awareness of the rapidly increasing importance of computer software systems in many activities of society. Thus, although much of the conference was concerned with detailed technical questions, many of the discussions were of a more general nature, and should be of interest to a wide spectrum of readers. It is for the benefit of this wider audience that representative discussions of various points relating to the impact of software engineering on society have been abstracted from later sections of this Report, and collected in this introductory section.

First, three quotations which indicate the rate of growth of software:

Helms: In Europe alone there are about 10,000 installed computers — this number is increasing at a rate of anywhere from 25 per cent to 50 per cent per year. The quality of software provided for these computers will soon affect more than a quarter of a million analysts and programmers.

David: No less a person than T.J. Watson said that OS/360 cost IBM over $50 million dollars a year during its preparation, and at least 5000 man-years' investment. TSS/360 is said to be in the 1000 man-year category. It has been said, too, that development costs for software equal the development costs for hardware in establishing a new machine line.

d'Agapeyeff: In 1958 a European general purpose computer manufacturer often had less than 50 software programmers, now they probably number 1,000-2,000 people; what will be needed in 1978?

Yet this growth rate was viewed with more alarm than pride.

David: In computing, the research, development, and production phases are 16 often telescoped into one process. In the competitive rush to make available the latest techniques, such as on-line consoles served by time-shared computers, we strive to take great forward leaps across gulfs of unknown width and depth. In the cold light of day, we know that a step-by-step approach separating research and development from production is less risky and more likely to be successful. Experience indeed indicates that for software tasks similar to previous ones, estimates are accurate to within 10–30 percent in many cases. This situation is familiar in all fields lacking a firm theoretical base. Thus, there are good reasons why software tasks that include novel concepts involve not only uncalculated but also uncalculable risks.

This is not meant to indicate that the software field does not have its successes.

Hastings: I work in an environment of some fourteen large installations using OS/360. These are complex systems, being used for many very sophisticated applications. People are doing what they need to do, at a much lower cost than ever before, and they seem to be reasonably satisfied.

Buxton: Ninety-nine percent of computers work tolerably satisfactorily. There are thousands of respectable Fortran-oriented installations using many different machines, and lots of good data processing applications running steadily.

However, there are areas of the field which were viewed by many participants with great concern.

Kolence: The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time. There are many areas where there is no such thing as a crisis — sort routines, payroll applications, for example. It is large systems that are encountering great difficulties. We should not expect the production of such systems to be easy.

David and Fraser: Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death.

Dijkstra: The dissemination of knowledge is of obvious value — the massive dissemination of error-loaded software is frightening.

17

There was general agreement that 'software engineering' is in a very rudimentary stage of development as compared with the established branches of engineering.

McIlroy: We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries.

Kolence: Programming management will continue to deserve its current poor reputation for cost and schedule effectiveness until such time as a more complete understanding of the program design process is achieved.

Fraser: One of the problems that is central to the software production process is to identify the nature of progress and to find some way of measuring it. Only one thing seems to be clear just now. It is that program construction is not always a simple progression in which each act of assembly represents a distinct forward step and that the final product can be described simply as the sum of many sub-assemblies.

Graham: Today we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes — build the whole thing, push it off the cliff, let it crash, and start over again.

Of course any new field has its growing pains:

Gillette: We are in many ways in an analogous position to the aircraft industry, which also has problems producing systems on schedule and to specification. We perhaps have more examples of bad large systems than good, but we are a young industry and are learning how to do better.

Many people agreed that one of the main problems was the pressure to produce even bigger and more sophisticated systems.

Opler: I am concerned about the current growth of systems, and what I expect is probably an exponential growth of errors. Should we have systems of this size and complexity? Is it the manufacturer's fault for producing them or the user's for demanding them? One shouldn't ask for large systems and then complain about their largeness.

18

Buxton: There are extremely strong economic pressures on manufacturers, both from users and from other manufacturers. Some of these pressures, which are a major contributory cause of our problems, are quite understandable. For example, the rate of increase of air traffic in Europe is such that there is a pressing need for an automated system of control.

This being the case, perhaps the best quotation to use to end this short section of the report is the following:

Gill: It is of the utmost importance that all those responsible for large projects involving computers should take care to avoid making demands on software that go far beyond the present state of technology) unless the very considerable risks involved can be tolerated.

Appendix B

21

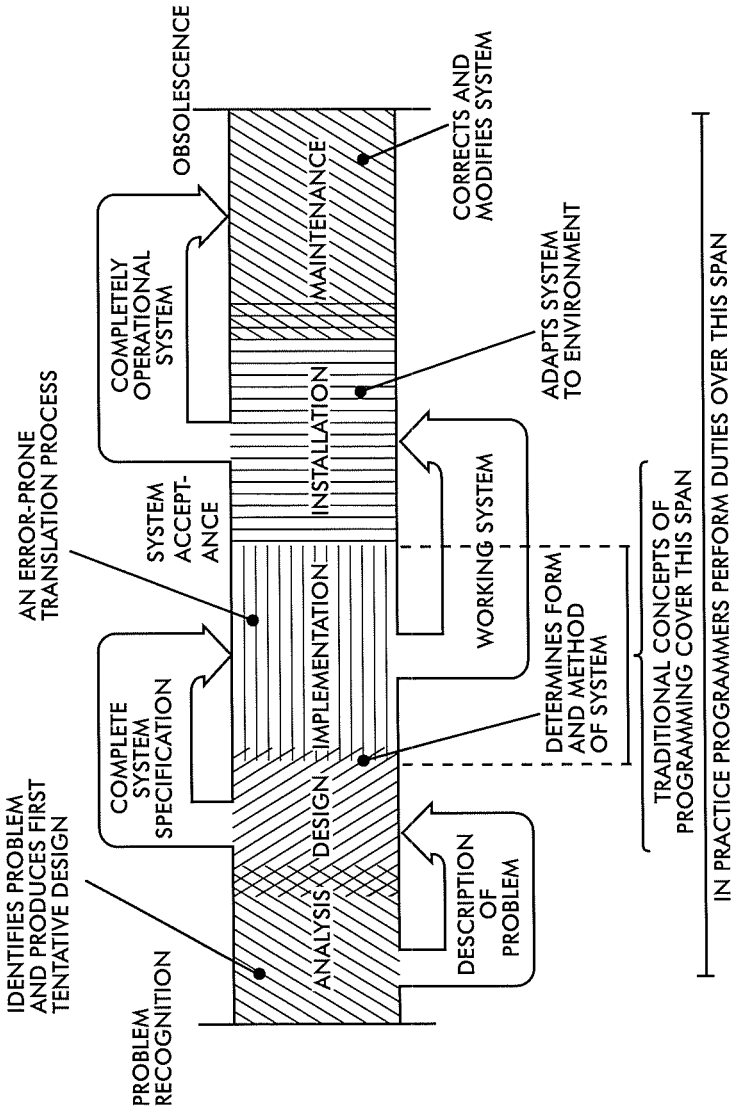

Figure 2. From Selig: Documentation for service and users. Originally due to Constantine.

Bibliography

Aho, A. V., and Ullman, J. D. (1979) *Principles of Compiler Design*, Reading, MA: Addison-Wesley.

Aho, A. V., Lam, M. S., Sethi, R, and Ullman, J. D. (2007) *Compilers. Principles, Techniques, and Tools*, London and New York: Pearson and Addison-Wesley.

Alt, F. L., and Rubinoff, M. (eds) (1968) *Advances in Computers*, New York: Academic Press.

Aristotle (1984) *The Complete Works*, Princeton, NJ: Princeton University Press.

Austin, J. L. (1972) *How to Do Things with Words*, Oxford: Clarendon Press.

Barad, K. (2003) 'Posthumanist Performativity: Toward an Understanding of How Matter Comes to Matter', *Signs: Journal of Women in Culture and Society*, 28 (3): 801-831.

Beardsworth, R. (1995) 'From a Genealogy of Matter to a Politics of Memory: Stiegler's Thinking of Technics', *Tekhnema* 2: non-pag., http://tekhnema.free.fr/2Beardsworth.htm.

Beardsworth, R. (1996) *Derrida and the Political*, New York: Routledge.

Belady, L. A., and M. M. Lehman (1976) 'A Model of Large Program Development', *IBM Systems Journal* 15 (3): 225-252.

Bolter, J. D. (1984) *Turing's Man: Western Culture in the Computer Age*, London: Duckworth.

Bolter, J. D. (2001) *Writing Space: Computers, Hypertext, and the Remediation of Print*, Mahwah, NJ and London: Lawrence Erlbaum Associates.

Bolter, J. D., and Grusin, R. (2002) *Remediation: Understanding New Media*, Cambridge, MA: MIT Press.

Braudel, Fernand (1973) *Capitalism and Material Life 1400-1800*, London: Weidenfeld and Nicolson.

Brooks, F. P. (1987) 'No Silver Bullet: Essence and Accidents of Software Engineering', *IEEE Computer*, 20 (4): 10-19.

Brooks, F. P. (1995) *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Reading, MA: Addison-Wesley, 1995.

Burrell, M. (2004) *Fundamentals of Computer Architecture*, Basingstoke and New York: Palgrave Macmillan.

Butler, J. (1997) *Excitable Speech. A Politics of the Performative*, New York and London: Routledge.

Buxton, J. N., Naur, P., and Randell, B. (eds) (1976) *Software Engineering: Concepts and Techniques*, New York: Petrocelli-Charter.

Buxton, J. N., and Randell, B. (eds) (1970) *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*, Birmingham: NATO Science Committee.

Callan, R. (2003) *Artificial Intelligence*, Basingstoke: Palgrave Macmillan.

Ceruzzi, P. (1989) *Beyond the Limits: Flight Enters the Computer Age*, Cambridge, MA: MIT Press.

Chomsky, N. (1956) 'Three Models for the Description of Language', *IRE Transactions on Information Theory*, 2(3): 113-124.

Chomsky, N. (1957) *Syntactic Structures*, The Hague: Mouton.

Chomsky, N. (1959) 'On Certain Formal Properties of Grammars', *Information and Control*, 2(29): 137-167.

Chomsky, N. (1965) *Aspects of the Theory of Syntax*, Cambridge, MA: MIT Press.

Chun, W. H. K. (2003) 'On the Persistence of Visual Knowledge'. Paper presented at the annual convention of the Modern Language Association, San Diego, CA, December 28.

Clark, T. (2000) 'Deconstruction and Technology', in N. Royle (ed.) *Deconstructions. A User's Guide*, Basingstoke: Palgrave, 238-257.

Clocksin, W. F., and Mellish, C.S. (2003) *Programming in Prolog*, New York: Springer-Verlag.

Constantine, L. L. (1968) 'The Programming Profession, Programming Theory, and Programming Education', *Computers and Automation* 17(2): 14-19.

Conway, M. E. (1968) 'How Do Committees Invent?', *Datamation* 14 (4): 28-31.

Cox, B. J. (1990) 'There Is a Silver Bullet'. *BYTE Magazine*, October: 209-18.

Culler, J. (1986) *Ferdinand de Saussure*, New York: Cornell University Press.

Dahl, O.-J., Dijkstra E. W., and Hoare, C. A. R. (1972) *Structured Programming*, London: Academic Press.

Derrida, J. (1973) *Speech and Phenomena and Other Essays on Husserl's Theory of Signs*, Evanston: Northwestern University Press.

Derrida, J. (1976) *Of Grammatology*, Baltimore: The Johns Hopkins University Press.

Derrida, J. (1978) *Edmund Husserl's Origin of Geometry: An Introduction*, New York: Nicolas Hays.

Derrida, J. (1979) *Spurs: Nietzsche's Styles*, Chicago: University of Chicago Press.

Derrida, J. (1980) 'Structure, Sign, and Play in the Discourse of the Human Sciences', in *Writing and Difference*, London: Routledge: 278-294.

Derrida, J. (1981) 'Plato's Pharmacy', in *Dissemination*, Chicago, IL: University of Chicago Press: 63-171.

Derrida, J. (1982) *Margins of Philosophy*, Chicago: Chicago University Press.

Derrida, J. (1983) 'The Principle of Reason: The University in the Eyes of Its Pupils', *Diacritics* (Fall): 6-20.

Derrida, J. (1985) 'Letter to a Japanese Friend', in R. Bernasconi and D. Wood (eds) *Derrida and Différance*, Warwick: Parousia Press: 1-5.

Derrida, J. (1986) *Mémoires: for Paul de Man*, New York: Columbia University Press.

Derrida, J. (1987) *The Post Card: From Socrates to Freud and Beyond*, Chicago: University of Chicago Press.

Derrida, J. (1988) *Limited Inc.*, Evanston: Northwestern University Press.

Derrida, J. (1989) 'Psyche: Inventions of the Other', in L. Waters and W. Godzich (eds) *Reading de Man Reading*, Minneapolis: University of Minnesota Press: 25-65.

Derrida, J. (1994) *Specters of Marx: The State of the Debt, the Work of Mourning, And the New International*, New York and London: Routledge.

Derrida, J. (1995) *Points...: Interviews, 1974-1994*, Stanford, CA: Stanford University Press.

Derrida, J. (1996) *Archive Fever: A Freudian Impression*, Chicago: University of Chicago Press.

Derrida, J. and Stiegler, B. (2002) *Echographies of Television: Filmed Interviews*, Cambridge: Polity Press.

Derrida, J. (2004) *Positions*, London and New York: Continuum.

Derrida, J. (2005) *Paper Machine*, Stanford, CA: Stanford University Press.

Dijkstra, E. W. (1968) 'Go To Statement Considered Harmful', *Communications of the ACM* 11(3): 146-148.

Doyle, R. (2003) *Wetwares: Experiments in Postvital Living*, Minneapolis: University of Minnesota Press.

DuGay, P., Hall, S., Janes, L., Mackay, H., and Negus, K. (1997) *Doing Cultural Studies: The Story of the Sony Walkman*, London: Sage/The Open University.

Eckel, B. (1995) *Thinking in C++*, Englewood Cliffs, NJ: Prentice Hall.

Fischer, C. N., LeBlanc, R. J. (1988) *Crafting a Compiler*, Menlo Park, CA: Benjamin/Cummings.

Fuller, M. (2003) *Behind the Blip: Essays on the Culture of Software*, New York: Autonomedia.

Galler, B. A. (1989) 'Thoughts on Software Engineering', *Proceedings of the 11th International Conference on Software Engineering*, 11 (2): 97.

Galloway, A. (2004) *Protocol: How Control Exists after Decentralization*, Cambridge, MA: MIT Press.

Gelb, I. J. (1963) *A Study of Writing*, Chicago: University of Chicago Press.

Gell, A. (1992) 'The Technology of Enchantment and the Enchantment of Technology', in J. Coote, and A. Shelton (eds) *Anthropology, Art, and Aesthetics*, Oxford: Clarendon Press: 40-63.

Gell, A. (1998) *Art and Agency*, Oxford: Clarendon Press.

Gere, C. (2003) 'Can Art History Go on Without a Body?', *Culture Machine* 5: non-pag., http://www.culturemachine.net/.

Gille, B. (1986) *History of Techniques*, New York: Gordon.

Glass, R. L. (1988) 'Glass' (column), *System Development*, January: 4-5.

Glass, R. L. (2003) *In the Beginning: Recollections of Software Pioneers,* Hoboken, NJ: John Wiley.

Gordon, R. M. (1968) *Review of 'The Management of Computer Programming Projects' by C.P. Lecht*, Datamation 14 (4): 11.

Gries, D. (1989) 'My Thoughts on Software Engineering in the Late 1960s', *Proceedings of the 11th International Conference on Software Engineering*, 11 (2): 98.

Habermas, J. (1991) *The Theory of Communicative Action*, Cambridge: Polity Press.

Habermas, J. (2001) *The Postnational Constellation*, Cambridge: Polity Press.

Habermas, J. (2003) *The Future of Human Nature*, Cambridge: Polity Press.

Halberstam, J. (2007) 'Pixarvolt: Animation and Revolt', *Flow Journal* 6 (6),: non-pag., http://flowtv.org/?p=739.

Hall, G. (2002) *Culture in Bits: The Monstrous Future of Theory*, London and New York: Continuum.

Hall, G. (2007a) 'The Singularity of New Media', paper presented at the 'Re-Mediating Literature' Conference, University of Utrecht, 4-6 July.

Hall, G. (2008) *Digitize This Book! The Politics of New Media, or Why We Need Open Access Now,* Minneapolis: University of Minnesota Press.

Hall, G. (2007b) 'IT, Again: How to Build an Ethical Virtual Institution', in S. M. Wortham and G. Hall (eds) *Experimenting: Essays with Samuel Weber,* New York: Fordham University Press: 116-140.

Hall, S. (1992) 'Cultural Studies and its Theoretical Legacies', in L. Grossberg, C. Nelson and P. Treichler (eds) *Cultural Studies*, New York: Routledge: 277-294.

Hall, S. (ed.) (1997) *Representation: Cultural Representations and Signifying Practices*, London: Sage/The Open University.

Hansen, M. (2003) '"Realtime Synthesis" And The Différance of The Body: Technocultural Studies In The Wake Of Deconstruction", *Culture Machine* 5: non-pag., http://www.culturemachine.net/.

Hansen, M. (2004) *New Philosophy for New Media*, Cambridge, MA: MIT Press.

Harel, D. (1992) 'Biting the Silver Bullet: Toward a Brighter Future for System Development' IEEE Computer, *25 (1): 8-20.*

Harry, H. and Minsky, M. (1992) *The Turing Option*, London: ROC.

Hayles, K. N. (1999) *How We Became Posthuman: Virtual Bodies in Cybernetics, Literature and Informatics*, Chicago: University of Chicago Press.

Hayles, K. N. (2002) *Writing Machines*, Cambridge, MA: MIT Press

Hayles, K. N. (2005) *My Mother Was a Computer: Digital Subjects and Literary Texts,* Chicago: University of Chicago Press.

Heidegger, M. (1977) *The Question Concerning Technology and Other Essays*, New York: Harper and Row.

Hicks, H. T. (1968) 'Modular Programming in COBOL', *Datamation* 14 (5): 50-59.

Hood, Webster J. (1983) 'The Aristotelian Versus the Heideggerian Approach to the Problem of Technology', in C. Mitcharn and R. Mackey (eds) *Philosophy and Technology: Readings in the Philosophical Problems of Technology*, London: The Free Press: 347-63.

Hopcroft, J. E., and Ullman, J. D. (1969) *Formal Languages and Their Relation to Automata*, Reading, MA: Addison-Wesley.

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001) *Introduction to Automata Theory, Languages, and Computation*, Reading, MA: Addison-Wesley.

Hopkins, M. (1968) 'SABRE PL/I', *Datamation* 14 (12): 35-38.

Humphrey, W. (1989a) *Managing the Software Process*, Harlow: Addison-Wesley.

Humphrey, W. (1989b) 'The Software Engineering Process: Definition and Scope', *Representing and Enacting the Software Process: Proceedings of the 4th International Software Process Workshop*, ACM Press: 82-83.

Husserl, E. (1970) *The Crisis of European Sciences and Transcendental Phenomenology. An Introduction to Phenomenological Philosophy*, Evanston: Northwestern University Press.

Jones, C. (2007) *Estimating Software Costs*, New York: McGraw-Hill.

Kaaranen, H., Ahtiainen, A., Laitinen, L., Naghian, S., and Niemi, V. (2005) *UMTS Networks: Architecture, Mobility and Services*, Chichester: John Wiley.

Kirschenbaum, M. G. (2002a) 'Editing the Interface: Textual Studies and First Generation Electronic Objects', in W. Speech Hill and E. M. Burns (eds), *TEXT: An Interdisciplinary Annual of Textual Studies*, 14, Ann Arbor: University of Michigan Press: 15-51.

Kirschenbaum, M. G. (2002b) 'Materiality and Matter and Stuff: What Electronic Texts Are Made Of', Electronic Book Review 12: non-pag., http://www.altx.com/ebr/riposte/rip12/rip12kir.htm.

Kirschenbaum, M. G. (2004) 'Virtuality and VRML: Software Studies after Manovich', in M. Bousquet and K. Wills (eds) *The Politics of Information: The Electronic Mediation of Social Change*, Alt-X Press eBook: non-pag., http://www.altx.com/ebooks/infopol.htm.

Kittler, F. A. (1997) *Essays: Literature, Media, Information Systems*, Amsterdam: G+B Arts.

Kittler, F. A. (1999) *Gramophone, Film, Typewriter*, Stanford, CA: Stanford University Press.

Kittler, F. A. (1995) 'There Is No Software', *CTheory*: non-pag., http://www.ctheory.net/articles.aspx?id=74.

Latour, B. (1993) *We Have Never Been Modern*, Cambridge, MA: Harvard University Press.

Leroi-Gourhan, A. (1993) *Gesture and Speech*, Cambridge: MIT Press.

Leavitt, D. (2006) *The Man Who Knew Too Much: Alan Turing and the Invention of the Computer*, New York: Weidenfeld & Nicolson.

Licklider, J. C. R. (1969) 'Underestimates and Overexpectations', in A. Chayes and J. B. Wiesner (eds) *ABM: An Evaluation of the Decision to Deploy an Anti-Ballistic Missile System*, New York: Signet: 118-129.

Lister, M., Dovey, J., Giddings, S., Grant, I., Kieran, K. (2003) *New Media: A Critical Introduction*, London and New York: Routledge.

Mackenzie, A. (2003) 'The Problem of Computer Code: Leviathan or Common Power?', non-pag.,http://www.lancs.ac.uk/staff/mackenza/papers/code-leviathan.pdf.

Mahoney, M. S. (2004) 'Finding a History for Software Engineering', *Annals of the History of Computing* 26 (1): 8-19.

Manovich, L. (2002) *The Language of New Media*, Cambridge, MA: MIT Press.

Manovich, L. (2008) *Software Takes Command*, http://lab.softwarestudies.com/2008/11/softbook.html.

Mateas, M. (2001) 'Expressive AI: A Hybrid Art and Science Practice', *Leonardo: Journal of the International Society for Arts, Sciences and Technology*, 34 (2): 147-153.

Marvin, C. (1988) *When Old Technologies Were New: Thinking About Electric Communication in the Late Nineteenth Century*, New York and Oxford: Oxford University Press.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, J. P., and Levin, M. (1965) *LISP 1.5 Programmer's Manual*, Cambridge, MA: MIT Press.

McCorduck, P. (2004) *Machines Who Think*, Natick, MA: A K Peters.

Meyrowitz, J. (1985) *No Sense of Place: The Impact of Electronic Media on Social Behavior,* New York: Oxford University Press.

Mills, H. (1971) 'Chief Programmer Teams, Principles, and Procedures', *IBM Federal Systems Division Report FSC 71-5108*, Gaithersburg, Md..

Mitchell, R., and Thurtle, P. (eds) (2004) *Data Made Flesh: Embodying Information*, New York and London: Routledge.

Morowitz, H. (2002) *The Emergence of Everything: How the World Became Complex*, New York: Oxford University Press.

Morrison, P., and Morrison, E. (eds) (1961) *Charles Babbage and His Calculating Engines: Selected Writings by Charles Babbage and Others*, New York: Dover.

Naur, P., and Randell, B. (eds) (1969) *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11st October 1968*, Brussels (Belgium): NATO Scientific Affairs Division.

Nilsson, N. T. (1998) *Artificial Intelligence. A New Synthesis*, San Francisco, CA: Morgan Kaufmann.

Ong, W. J. (1971) *Rhetoric, Romance and Technology: Studies in the Interaction of Expression and Culture*, Ithaca, NY: Cornell University Press.

Ong, W. J. (1982) *Orality and Literacy: The Technologising of the Word*, London and New York: Routledge.

Parnas, D. (1972) 'A Technique for Software Module Specification with Examples', *ACM Communications* 15(5): 330-336.

Parry, Richard (2003) '*Episteme* and *Techne*', *The Stanford Encyclopedia of Philosophy (Summer 2003 Edition)*, E. N. Zalta (ed.), http://plato.stanford.edu/archives/sum2003/entries/episteme-techne/: non-pag.

Pinney, N., and Thomas, N. (eds) (2001) *Beyond Aesthetics: Art and the Technologies of Enchantment*, Oxford and New York: Berg.

Plant, S. (1995) 'The Future Looms: Weaving Women and Cybernetics', in M. Featherstone and R. Burrows (eds) *Cyberspace/Cyberbodies/Cyberpunk: Cultures of Technological Embodiment*, London: Sage: 45-64.

Plant, S. (1998) *Zeros and Ones: Digital Women and the New Technoculture*, London: Fourth Estate.

Plato (1989) *The Collected Dialogues*, Princeton, NJ: Princeton University Press.

Plato (2000) *The Republic*, Cambridge: Cambridge University Press.

Poster, M. (2002) 'High-tech Frankenstein, or Heidegger Meets Stelarc', in J. Zylinska (ed.) *The Cyborg Experiments: the Extensions of the Body in the Media Age,* London and New York: Continuum: 15-32.

Poster, M. (2006) *Information Please: Culture and Politics in the Age of Digital Machines*, Durham: Duke University Press.

Raley, R. (2003) 'Machine Translation and Global English', *Yale Journal of Criticism*, 16 (2): 291-313.

Randell, B. (1979) 'Software Engineering in 1968', *Proceedings of the IEEE 4th International Conference on Software Engineering*, Munich: 1-10.

Randell, B. (1998) 'Memories of the NATO Software Engineering Conferences', IEEE Annals of the History of Computing 20(1): 51-54.

Raymond, E. S. (2000) 'The Cathedral and the Bazaar', *First Monday* 3(3): non-pag.,http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/578/499.

Reynolds, R. (1994) *Super Heroes: A Modern Mythology*, Jackson, MS: University Press of Mississippi.

Ross, D. (1989) 'The Nato Conferences from the Perspective of an Active Software Engineer', *Proceedings of the 11th International Conference on Software Engineering*, 11 (2): 101-2.

Salomaa, A. (1973) *Formal Languages*, London: Academic Press.

de Saussure, F. (1988) *Course in General Linguistics*, Peru, IL: Open Course Publishing.

Scarry, E. (1994) *Resisting Representation*, Oxford: Oxford University Press.

Sebesta, R. W. (2008) *Concepts of Programming Languages*, London and New York: Pearson and Addison-Wesley.

Sedgwick, E. K. (2003) *Touching Feeling: Affect, Pedagogy, Performativity*, Durham and London: Duke University Press.

Shaw, M. (1989) 'Remembrances of a Graduate Student', *Annals of the History of Computing, Anecdotes Department,* 11 (2): 141-143.

Shaw, M. (1990) 'Prospects for an Engineering Discipline of Software', *IEEE Software*, 7 (6): 15-24.

Simondon, G. (1989) *L'Individuation psychique et collective*, Paris: Aubier.

Simondon, G. (2001) *Du mode d'existence des objects techniques*, Paris: Aubier.

Sommerville, I. (1995) *Software Engineering*, Harlow: Addison-Wesley.

Stiegler, B. (1996) *La technique et le temps 2: La désorientation*, Paris: Galilée.

Stiegler, B. (1998a) *Technics and Time, 1: The Fault of Epimetheus*, Stanford, CA: Stanford University Press.

Stiegler, B. (1998b) 'The Time of Cinema: On the "New World" and "Cultural Exception"', *Tekhnema: Journal of Philosophy and Technology* 4: 62-118.

Stiegler, B. (2001a) *La Technique et le temps 3: Le temps du cinéma et la question du mal-être*, Paris: Galilée.

Stiegler, B. (2001b) 'Derrida and Technology: Fidelity at the Limits of Deconstruction and the Prosthesis of Faith', in T. Cohen (ed.) *Jacques Derrida and the Humanities: A Critical Reader*, Cambridge: Cambridge University Press: 238-70.

Stiegler, B. (2003a) 'Our Ailing Educational Institutions: The Global Mnemotechnical System', *Culture Machine* 5: non-pag., http://www.culturemachine.net/.

Stiegler, B. (2003b) 'Technics of Decision: An Interview', *Angelaki* 8 (2): 151-168.

Tanenbaum, A. S. (1999) *Structured Computer Organization*, Englewood Cliffs, NJ: Prentice-Hall.

Turing, A. (1950) 'Computing Machinery and Intelligence', *Mind*, 59(236): 433-460.

Ullman, E. (1997) *Close to the Machine: Technophilia and Its Discontents*, San Francisco, CA: City Lights Books.

Virilio, P. (1994) *The Vision Machine*, Bloomington: Indiana University Press.

Virilio, P. (1997) *Open Sky*, London: Verso.

Williams, R. (1961) *The Long Revolution*, Harmondsworth: Penguin.

Wirth, N. (1976) *Algorithms + Data Structures = Programs*, Englewood Cliffs, NJ: Prentice Hall.

Wolfram, S. (2002) *A New Kind of Science*, New York: Wolfram Media.

Filmography

Monsters vs Aliens (2009), directed by Rob Letterman and Conrad Vernon.

The Ister (2004), directed by David Barison and Daniel Ross.

The Net: The Unabomber, LSD and the Internet (2003), directed by Lutz Dammbeck.

The Trap: What Happened to Our Dream of Freedom (2007), directed by Adam Curtis.