Kunsthochschule für Medien Köln
Fächergruppe Kunst- und Medienwissenschaften

Open collaboration practices in software culture and their impact on the networked society

Dissertation
zur Erlangung des akademischen Grades Dr. phil.
im Fach Kunst- und Medienwissenschaft

Vorgelegt von
Luis Fernando Medina Cardona
aus Bogotá, Kolumbien

Prof. Dr. Georg Trogemann
Kunsthochschule für Medien Köln

Prof. Dr. Frieder Nake
Universität Bremen

Verteidigt am 14.01.2021

Some of the contents of this dissertation have already been published in other forms. The relevant publications are listed below.

En código abierto (in Open Source). *Arcadia Magazine "mutantes digitales" (Digital Mutants)*, 107, p. 43. Bogotá, 2014.

Exploraciones iniciales sobre la colaboración, el software libre y/o de código abierto y su impacto en la sociedad (Initial approaches to collaboration, free/open source software, and their impact on society). *Memorias del Festival Internacional de la Imagen (Proceedings of the International Image Festival).* Manizales, 2015.

La colaboración en la cultura del software: presentación del problema, resultados intermedios y exploración inicial de un estudio de caso en la cultura "maker" (Collaboration in the software culture: presentation of the issue, mid results, and initial approach to the "maker culture" study case.). Memorias del Encuentro Nacional de Investigación en Posgrados ENID (Proceedings of the National Conference of Postgraduate Research). Manizales, 2015.

The Free/Open Source Software as a Hybrid Concept: Considerations from a Colombian Activist's Perspective. *Media-N Journal of the New Media Caucus, "Mestizo Technology: art, design and technoscience in Latin America".* Vol. 12, N. 1, p. 76-81. 2016.

Exploraciones sobre la cultura del software libre o abierto como metáfora de la colaboración en la sociedad en red (Exploring the free/open software culture as a metaphor of collaboration in the network society). *Actio: Journal Of Technology in Design, Film Arts and Visual Communication (2)*, p. 114-125. 2018.

Open collaboration practices in software culture and their impact on the networked society

Abstract

This PhD dissertation addresses the open and collaborative mode of production in software. Specifically, it examines how various practices in the software culture have evolved and their relevance in the construct of the network society. It begins with a philosophical discussion in which a modern philosophy of technology points to technology as a system of thought and software as a technical culture. Not unlike the open and collaborative mode of production, software is source of metaphors. Upon these foundations, it undertakes the evolution of open practices from a historical and structural position. The historical account follows the premise that open collaborative practices of software precede the well-known Free/Libre Open Source Software (FLOSS). It presents some examples, first, related to the history of software and then to computer networks to track the motives and transformation of the open collaboration metaphor. The structural approach presents modern open collaboration in software as the result of a sociotechnical network composed of actants (node/artifacts), executed, in turn, by a collective made up of a human community and technical developments, in which textual machines for coding and communication are highlighted. Finally, the conclusion posits the findings and three modes of agency in software (algorithmic, interactive, and distributive). It also and suggests hybridization as the means to overcomes some shortcomings of the software open metaphor rhetoric.

Contents

1 — Introduction

1.1 Research context

We are living in the age of software. This statement -which may have formerly sounded bold- can now be made based on simple inspection of our everyday surroundings; the state of society, business operations, work, education, and leisure are examples that show us that it is so. Although this statement appears very familiar, it is only one among many other descriptors of similar significance, such as computer age, digital age, and network age. It may also bring to mind old cultural memes from science fiction and popular culture, such as space-age or atomic age. No matter what our perception of this statement is, one thing is clear, it is important enough to raise the interest of journalists, scholars, and creators, as well as of the community at large. It is relevant and powerful enough to be a metaphor for one of the most noticeable features of a particular period of human civilization. However, it is my belief that software is a precise descriptor of the current state of our society. There are other related terms like *computer*, which is now strongly connected to a commodity, a trend and no longer at the core of Business, as the shift from hardware to software in Industry has shown [Campbell-Kelly, 2004]. *Digital*, on the other hand, despite its popularity to describe binary electronic technologies, is a term that has been depleted of meaning by marketing, let alone the inaccuracy which it embodies [Cramer, 2014][1]. It would be fair to say that *network* is a fine descriptor, given the rise of the information exchange-based network society; in other words, computer networks. Although there is some truth in this affirmation, it could be argued that software is the central gear that moves the networks' infrastructure and, at the same time, empowers its connecting interfaces[2]. The importance of

[1] As it is noted in the reference; digital refers merely to a discrete numerical representation of phenomena, where electronic binary codification is just one case among several possibilities.

[2] The expression "networked software", proposed by Berry [Berry, 2011b, 2], takes this symbiotic relationship into account.

software should be clear after this short discussion.Although there may be several approaches, the assessment of modern society shall meet somewhere with the knowledge of software as a complex whole.

Indeed, at some stage, most people will engage in activities related to the input, processing, or results of a software-controlled task. In some instances, these activities are deliberate, and the person is completely aware of the software application, its interfaces, and how to enter the commands to undertake some activity (most of the time, the user can recognize the regular paths of software interaction despite not knowing what happens inside,). However, there is also software that runs in a seemingly invisible way, as in traffic lights or heating systems, unnoticed by the common person. Even in the most extreme scenarios, where people are computer illiterate or detached from technologically driven modern life, the chances are very high that their lives are affected by software-related tasks, with their data residing in some obscure database or as subjects of a governmental policy based on a software model. Simply put, if a person today is not interacting somehow with software, or is not indirectly affected by it, the possibility that this person lives entirely outside the modern global society is very high [Kitchin and Dodge, 2011]. As a result, software as a medium, which in turn can simulate several other media [Manovich, 2013], has become a strong academic point of interest that demands transdisciplinary scholarly approaches. Nevertheless, it remains a challenging subject to approach, given its highly technical nature and the tendency to focus only on its impacts, leaving its inner logics undiscovered.

This last observation may seem contradictory concerning modern society and software, that is, that even though *software* is all around us and even carries our conversations, there is little idea of what it is, precisely[Chun, 2011, 2]. Looking more closely at some aspects of this statement, we can state that the importance of software has significantly increased for the everyday person given political and economic events that have been news in the last years. The Snowden case, for example, and issues concerning online privacy, affecting governments and corporations, have somehow increased awareness of the risks of software mediated communication [Metahaven, 2015, 43,44]. The so-called "Arab Spring", around the turn of the second decade of the 21st century, awoke a global wave of techno-utopianism that celebrated the new possibilities afforded by social media in provoking social change[Kluitenberg, 2011, 7]. The startup centric economic discourse of several governments has brought to the fore a particular way of creating value out of software harvested information processed in real-time, following the Silicon Valley

model[Zuboff, 2015].

A long list, which involves mostly regular conversations (no matter if well understood), supports the perception of software as a mediator. It is here where the relevance of software as a Metamedium lies. In his frequently cited, "The Language of New Media", Manovich argues that people's familiarity with media, which he calls, "new media", rests on their media literacy, acquired through their relationship with other media. New media procedures resemble the operations of print, cinema, painting, and others [Manovich, 2002, 117]. Within media theory, however, this idea is not new but rather cliché. According to Mulder, "the computer, so goes an oft-repeated proposition, remediates all preceding media" [Mulder, 2004, 195]; this is also the main topic of an entire study by Bolter and Grusin [Bolter and Grusin, 2000]. This idea continues to be a central point in media and software theory, as expressed by terms such as "supermedium" [Berry, 2008], or even a more metaphorical term (with a la Turing amusements of the tandem computer-software), "Universal media machine" [Lunenfeld, 2011]. However, it should be understood that software is not only a "remediator"; it also has its own idiosyncrasies. Despite our reasonable familiarity with software mediations, interfaces, agencies, and affordances and its presence in our phones, appliances, offices, workplaces, and schools, we perceive the internal mechanisms of software as complex, obscure, and sometimes even irrelevant. This disregard makes software an object of study that provides a fertile field to pursue scholar undertakings to shed some light on its structure, behavior, and effects in general. It is not far-fetched to say that grasping the very essence of software allows us to assess our own identity in a networked software mediated world.

This project lies within the context of the software culture[3], and will address a particular aspect of the software realm, collaboration. Collaboration has come to the foreground in several discourses around software; it is an important part of current technosocial narratives such as the "sharing economy", and the "collaborative design". It is also at the core of the software metaphor used, which has spread into other domains. It embodies the idea of a read/write culture, and its role within the software culture requires an interdisciplinary approach.

[3]This term will be discussed in Chapter 2.

1.2 Research motives

An overview of several research motives is required before stepping into a properly formulated hypothesis on collaboration within the so-called software culture. Proposing collaboration as one central issue in the history of software and its relationship with other disciplines is, at the same time, a political statement and a move to harness the effects that the collaborative software-driven culture has unleashed through the tandem that computer-networks have had upon current software and new media narratives. In doing so, the comprehension of software will be enriched, contributing to a better understanding of the how and why the software metaphors (collaboration as one of the most important) have reached such a prominent place in the different fields where they are used (and abused). The assumptions being made in this work are informed by the dialog between the following general motivations.

First, the motivation to establish the role that software has within media or, more precisely, new media theory is essential. Is software a tiny parcel inside new media theory? Is it the most important issue? Can it even be considered equivalent? Several interesting issues arise regarding the relationship between software and new media theory. As mentioned in the previous section, the refashioning of prior media and the shaping of the modern world through computer code poses interesting questions about our society. Two approaches, with more than a decade between them, illustrate this concern. Manovich's explanation of what can be called new media is well known and used; it includes the media that displays the following five "principles": numerical representation, modularity, automation, variability, and transcoding. The focus of these properties is on media objects such as images, web pages, movies, and virtual reality, among others. All these examples are software mediated, an advantage, given their "numerical programmability" and the malleability of numerical representation [Manovich, 2002, 49]. Diversely, Federica Frabetti denounces, at the beginning of her "Software Theory", the problems of definition of the so-called "new media" (or new technologies). She takes it one step further by making a claim that makes software the common ground where this kind of media unfolds. She states, "[t]aking 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" [Frabetti, 2015, 3]. Even though both authors are dealing with the same topic and their positions are not very different, the simplicity of the second statement is a clear demonstration of the increasing

importance of software.

Another strong motivation to pursue the study of collaboration in software culture is FLOSS (Free/Libre Open Source Software). Numerous scholarly undertakings have encouraged the study of this kind of software. Particularly, those that consider direct access to code as a condition to conduct a well-informed analysis of computer applications, ranging from technical disciplines such as software engineering to more humanistic oriented approaches such as software studies [Fuller, 2008] or critical code studies [Berry, 2011b]. It should be clear that the properties of FLOSS enable its use in execution programs for every purpose and redistribution. That is, it offers the ability to inspect the internal code, make modifications upon it (code), and redistribute the changes[4], providing a significant advantage in understanding the software, starting with the code lines themselves. However, there is a more critical and ubiquitous motive; if software has become a metaphor to explain several phenomena, the properties of FLOSS are at the core of this conceptual transposition. A brief overview of several discourses shows an abundance of examples of the *openification* of several fields. There are free/open science, free/open culture, free/open politics, free/open art, and so on. This tendency, which started at the turn of the century, exemplifies the success of software metaphors. Hence, the relevance of searching for the causes, as well as the conveniences, inaccuracies, and problems of this selected angle to explain the surrounding reality; this can be considered as one instance of the "computational turn", which has led to such speculations and problematic statements considering "reality as a digital computer" [Mulder, 2004, 15]. Two central narratives have appeared to give this movement depth and legitimacy, perhaps motivated by FLOSS. One account presents FLOSS as a continuation of a venerable tradition of code sharing rooted in the hacker ethics dating back from the late fifties; however, this assessment presents several problems [Moody, 2002]. The other account claims that FLOSS is an unprecedented revolution that took the corporate software world by surprise [Frabetti, 2015]. This aspect must also be clarified here, which is why *collaboration* has been posited as the main narrative of this work[5]. It is also worth noting that the current FLOSS movement came into

[4]his short enouncement is based on the well-known "four freedoms" by Richard Stallman in 1984.

[5]There are other important aspects in software history, such as programming languages, methodologies, abstractions, and tools, among others. However, as stated, collaboration has had a political motivation to offer an alternative to the industry's competition trend. Collaboration also addresses human-human and human-machine interactions, as explained in the fifth chapter from a structural perspective.

being in the eighties and took off with the dissemination of the Internet in the nineties, which leaves plenty of historical terrain to understand software better. Consequently, there is a strong link between collaboration in software and computer networks that demands further examination, particularly considering the open nature of some networks' protocols.

Interdisciplinary studies also motivate this work. Literature in software studies firmly insists that software should be regarded from an integrated perspective, where different disciplines converge. Hence, several proposals encourage drawing from social sciences, computer science, cultural studies, digital humanities, semiotics, among others, to obtain a richer understanding of software[Frabetti, 2015, xi-xii][Manovich, 2013, 15]. This work follows this trend but begins with the following simple assessment: it is insufficient and always biased. Although this is standard and an objective approach would be impossible, there is still a need for more dialog between the technical disciplines. Software engineering, in particular, is often used superficially to support claims that, for the most part, are acceptable. However, they would provide a stronger argumentation and a more solid case if they were used with a more in-depth knowledge of the base, methods, techniques, and tools, in other words "the basic blocks", of software engineering[Goffey, 2008, 16]. In this line, the exploration of the XP (extreme programming) development methodology by Mackenzie [Mackenzie, 2006] can be mentioned or Taubert's approximation from sociology to the free software development process [Taubert, 2006]. In this work, we will employ a hybrid approach where software engineering (particularly collaborative and free software), digital humanities accounts, and computer history converge into the pursuit of the interdisciplinary ethos of software studies.

Lastly, there is also the motivation for grasping the concept of materiality. This topic has been largely discussed in the literature on software studies, but it still requires attention. Recently, there has been an ostensible theoretical agreement on the inutility of the immaterial/material dichotomy when discussing software; this frequent duple is described as belonging to the nineties (with the spreading of the Internet in the nineties and the preceding development of the so-called virtual reality technologies in the eighties). Nevertheless, this duple still permeates mass media and everyday perception of software, and a few scholarly accounts. The key issue here, of course, is to carefully analyze the different theoretical operations and arguments given to bridge the conceptual gap imposed by this misunderstanding. As stated in the introduction of one of the canonical works on software studies: "[w]hile this formulation [software alleged immateriality] has been deployed by many

writers to explain software's distinction from things that have a straightforward physicality at the scale of human visual perception or the way in which its production differs from industrial or craft forms of fabrication the idea of software's *immateriality* is ultimately trivializing and debilitating" [Fuller, 2008, 3]. This citation outlines how this work will present its reading on the immateriality/materiality dilemma, from the very material substrate of language operations to the importance of social operations, or even considering the physicality implied by the so-called "maker culture". In doing so, the theoretical discussion will not only inform the conclusions but will also provide several examples in which the distinction does not hold.

1.3 Hypothesis and Objectives

At this point in the discussion, the importance of software in our daily lives has been established. Similarly, the motivations described have provided a set of specific points of interest on the global idea of software. This work unfolds within this general framework, which offers a reasonably recognized background as a starting point. A direct top-down approach is used to clarify the real academic pursuit and state the research problem, the hypothesis, and the work's objectives. First, the working concept of "software culture" is needed (though its actual meaning will be explained throughout the work, especially in Chapter 2) as a construct that comprises the technical and social discourses surrounding computer applications from the design, programming, deployment, and everyday use of software. End users, systems analysts, programmers, and general stakeholders are considered as part of the actor-network-technology environment surrounding software (socio-technical network). The operation that motivates the main claim is simple; the so-called software culture also includes software metaphors. Plainly speaking, the rhetorical figures produced from software applied to another field (think of the image of the universe as a computer [Goffey, 2008, 16], running the code that we consider the reality described above)[6]. The increasing popularity of such a metaphor could be perceived as a sign of the so-called "computational turn"[Berry, 2011b, 23]. This work focuses mainly on the metaphor(s) related to FLOSS, where code (or the thing rendered as equivalent to code, depending on the field) is shared between peers and is used freely to be combined with new developments to build new things upon it. In order to pose a well-defined research undertaking, the word "collaboration" was selected as an aggregation of several practices in software development and considered

[6]This is one example of a doctrine called *computationalism*, where several things (the brain, society, for example) are portrayed as a computer [Berry, 2011b, 11,12].

in a broader context than the FLOSS movement. This step reports the advantage of thinking within the whole software culture frame, and not only a particular movement (despite the importance of FLOSS in the narrative). It is also necessary to consider this collaboration in the current network society without overlooking that the network idea itself has also become a recurrent metaphor in general media and academic descriptions.

Now, having described the general to the particular procedure above, and enouncing the key elements to understand the research question, a concrete question can be proposed. *How did the practices, methodologies, and artifacts that allow collaboration in the software development history evolve until they were shaped into the current free/open-source metaphor, and what are the implications of translating this metaphor and its working ethics into a non-software field?* Two relevant aspects emerge from this question; the need for a robust methodological system and the combination of a historical and structural approach. Along with these two aspects, we need to address other auxiliary questions that revolve around the objectives of this work and its main pursuit[7].

The main hypothesis is twofold. First, collaboration is not new to software development; thus, to software culture. Throughout the history of software, it has been present in various forms. Second, software is an unstable medium, consequently bringing about the instability concept to the free/open metaphor. Therefore, there have been several approaches to define this collaboration and other associated practices. Collaboration acquired its current form with the appearance of the FLOSS movement, coalescing with the free software movement's definitions, and later, the open-source movement. This current form embraces software instability through an ever-changing sociotechnical network.

Moreover, to bridge the software realm with the computer network's, it is argued that this highly interrelated platform (software/network tandem) was one of the first consummations of the free/open software-based metaphor, even before this metaphor was branded as such. With protocols, communications software, and design as forbearers of other metaphor incarnations, it is easy to transition from those elements to the mentioned tandem in modern software development. By analogy, if the software culture metaphors, including the central free/open metaphor, are applied to other fields, they

[7]Collaboration is preferred over cooperation. Its use is broader in the software realm, and it suits modern software development better, as explained in section 5.2.

would have their own instabilities related to the software's.

The following are the general objectives

- To critically question the construct of the software culture as an instrument to describe software from a broader perspective where its stakeholders, social effects, and technical particularities are discussed, using collaboration as the guiding concept.
- To synthesize the so-called free/open metaphor as one perceptible effect of the software culture in a post-digital society where the collaboration concept has acquired its current understanding and where it is possible to observe the non-deterministic, unstable nature of software.

The following specific objectives outline the methodological approach to achieve the general aims.

- To discuss the idea of software as a tool or a culture under the light of the human technicity concept.
- To build a genealogy of software evolution considering the social and technical aspects from a collaboration perspective.
- To discuss the free/open metaphor as a prominent example of the software metaphors driven by the increasing influence of computer applications in our daily lives.
- To present and analyze the software development collaboration support provided by different software tools, protocols, and software engineering techniques produced by the emergence of the FLOSS concept.
- To consider some of the implications of extending software collaboration tropes into other fields.
- To question the popular narrative of collaboration in software as homogeneous and provide insight on its diversity.

1.4 Methodological approach

The proposed methodology not only discusses a reference system that combines literature from a humanistic and technical background but also tackles how the perceived hype of FLOSS, from the 1990s, has changed and has been appropriated by several fields. Likewise, the proposed narrative seeks to put

collaborative software tropes in a broader retrospective from a historical and technical structural point of view. Through these two lenses, history and structure will be intertwined using a collaborative focus to discuss and problematize the evolution of collaborative modern software development and the modern structure of networked collaboration. The following methodological approach and focuses were used in this study:

- Critical cross-reading: as mentioned, the textual corpus to be built for this work is mainly drawn from humanistic and technical sources. Firstly, from media-related literature, which provides a broad framework in which software can be located within a general idea of technology and its definitions and relationship with society. The philosophy and sociology of technology, in general, come into play here, in particular of software, to discuss the perception of computer applications and code under the premise of media as a mechanism; secondly, from technical literature, namely, software engineering, which will be brought into dialog with the philosophical part. Both, to appraise the value of software products in supporting collaboration (version control systems, communications protocols, and devices, abstract artifacts, etc.) and sustain the humanistic account of software. It should be noted that there is an increasing interest in engaging with the technological discourses of software studies, as in [Frabetti, 2015] or [Mackenzie, 2006].

  Conversely, the contribution of humanities to software development, as observed from software engineering, remains difficult to assess. Therefore, cross-reading is almost mandatory in current scholarly work, considering the overwhelming availability of resources and the influence of hypertextual narratives on academic writing, let alone the software encouraged remix practices. *Critical reading* should also be further explained in this context. As pointed out previously, technical narratives sometimes describe software as a tool using a biased discourse that claims alleged neutrality and deterministic behavior. Indeed, some works denounce the inconvenience of the instrumental appraisal of software[Berry, 2011b][Frabetti, 2015]. Hence, the need for critical approaches to the text, depending on the desired perspective. Contrasting references must be used to offer a critical point of view on FLOSS, the software culture, and collaborative modes of production while simultaneously identifying problematic passages and framing them within the research context. This strategy is especially necessary regarding the extensive corpus of gray literature, such as websites, journalistic accounts, and activist reports, which are valuable sources of information

but prone to the misuse of terms and inaccuracies.

- The modular granularity of code: according to critical code studies [Marino, 2020], computer code and its analysis offer advantages to understanding software [8]. However, taking into account the reach of the objectives of this work, code studies are a specific part of software studies, suited to approach an algorithm or computer module than a complete mode of production; this means that code is considered a product. Indeed, it is an object of interest, but from the perspective of how it is conceived, shared, transmitted, managed, and delivered upon, etc. not solely from the computer language position. Many approaches have used similar strategies with relevant and well-structured outcomes, giving proper understanding to continuing the discussion on software materiality. However, here, granularity should be further explained. Different appraisals of code are necessary, resembling *zoom levels*, given the evolution of code and the appearance of modularity techniques. Consider this affirmation, "[a] code, in the sense that Protocol defines it, is process-based, it is parsed, compiled, procedural or object-oriented, and defined by ontology standards" [Galloway, 2004, XIII]. Clearly, agency of code (process-based) instructs the modern perceptions of code and ergo software. However, the important point here is the possibility of navigation across several layers of code granularity using programming and code organization frameworks such as structured and object-oriented programming. In doing so, collaboration is grasped through protocols, modules, programs, repositories, and other complex artifacts that are at stake when network-based software collaboration occurs.

- Iterative approach: this is another consequence of a software metaphor and one that displays the materiality that extends such metaphors, in this case, into the research structure. It is now commonly accepted that research is not a linear cumulative path. Instead, it is an undertaking full of obstacles, steps backward, reconsiderations, and even dramatic changes in objectives. Therefore, it makes sense to approach this research topic in this fashion, alternating reading with writing phases into an iterative form that moves towards the main purpose incrementally. It should be noted that reading and writing are not exclusively a phase. One of the advantages of cross reading is the potential of using snippets of a text on demand –*on the fly* – to support a particular claim

[8] In fact, as put by Berry, "code and software are two sides of the same coin" [Berry, 2011b, 32]

or one merely *discovered* while skimming through a text. Similarly, notes, ideas, and small argumentation constructs can be made while reading, which, using the proper indexation system, can be included in the main draft in the writing phase. An iterative approach does not dictate a fixed duration or end to an iteration, the necessary number of iterations can be made before the final draft or version. An iterative approach is flexible; it provides the ability to make small changes of direction, correct past mistakes, and re-organize the whole structure, according to subsequent feedback. Although this approach is not new, it intends to offer a more realistic plan than the one commonly used in modern theoretical scholarly works.

- Historical and structural approaches: these are the prevalent methodological approaches in this work. They refer not only to a particular point of view and way of organizing tasks (organization of literature) but to the writing structure itself. The research question and the corresponding hypothesis require the assessment of the evolution of software collaboration from its beginnings in computer history. To this end, it was necessary to delve, not only chronologically but also critically, into the different manifestations –technical and social– that can be considered an instance of collaborative modes of production in software development. Although one of the premises of this work is the non-linear conform of technology history, as detailed in the following chapter from the perspective of the philosophy of technology, the media archeology-based approach will evaluate, in an unconstrained order, different examples of collaboration as they appear in the technical literature. However, the discussion will be within the correct context to present the richness and temporally entangled nature of the collaboration narrative. This historical substrate will inform other sections organized following the idea of the software/network tandem. A more structural approach will be taken once the historical narrative and its importance are established. The implicit idea is that once the collaboration tropes are clear, modern open and collaborative modes of software production can be described as an assemblage of technical and social artifacts, interacting via different communication protocols to develop a piece of software using open collaboration ethics. The word network will be used to describe this structure, stressing on the mutable state of networks and their collaborative background. Ultimately, the historical and structural approaches are intertwined and provide a platform to discuss how open, collaborative motives evolved and how they work today.

It should be stated that methodologically speaking, in the beginning, there was a thought of conducting a field study using the so-called "maker culture" as a vehicle to observe and prove how the open and collaborative metaphor went beyond the software field. It contested the alleged immateriality through the physicality implied in software-based fabrication (using equipment such as CNC machines, 3D printers, etc.). After some iterations, this idea was discarded because –though a case study could be useful– it would reflect a deterministic perspective of the existence of a straight demonstration for the hypothesis on one side; this would require an entirely different effort and an entirely different work. Therefore, the maker culture is mentioned when necessary but is not a considerable part of this work.

1.5 Sources

Special attention must be given to the selected sources of information. It should be noted that a cross-reading approach obtains its information mostly from scholarly books and articles devoted to the topics addressed. However, software and software culture is a broad concept that covers an extension of material that sometimes surpasses the strict limits of academic reference material. Moreover, the computer networked public sphere encompasses a vast field of interests comprising texts, images, computer code, and other sources that contain valuable information. These sources can provide examples, empirical cases, historical facts, or particular pointers to a specific issue that reinforces the core of scholarly based claims. In other words, the main body will involve theoretical developments in the field using other documents to bolster the soundness of the arguments; this is almost obligatory when communities such as FLOSS are questioned. Thus, the following sources of information were considered:

- Journalistic accounts: the evident increase in press and media covering computer developments and software, as well as the social implications of these technologies, are clear examples of its essentialness in modern society. The information offered by this kind of media, albeit their accuracy, is a vivid illustration of the expectations and popular assumptions about technology. Magazines openly reflect the spirit of a particular age by using illustrations, pictures, and even humor. The stories covered by software magazines directed to the business person or enthusiast are valuable for this work, as they contain early references to topics such as collaboration, FLOSS, and community networking, among others.

The advantage of these texts is their free-form and the diversity of voices and interests. Two highly recognized sources can be cited to illustrate the importance of this media. The first source is the esteemed magazine, "Datamation", founded in 1957. Its long tradition has already been used as a source to track the evolution of certain software traits, for instance, in [Campbell-Kelly, 2004]. At the other end of the spectrum are publications like "2600 Quarterly", devoted to hacker issues. Including such sources reinforces the concept of "software culture" through important but sometimes neglected sources.

- Technical documents: although these documents are part of the standard reference system of software studies, it should be noted that they are insufficient. Software engineering documents and programming specifications are recurrently referenced concerning the origins of software engineering in [Frabetti, 2015], as well as for Java standards in [Mackenzie, 2006]. Besides the multidisciplinary perspective provided by computer science literature, there is a vast potential in documents such as manuals, how-to guides, release notes, etc., as well as the gray literature associated with software releases, especially FLOSS. Heeding the classic definition of software as not only the lines of code but also "the documents"[Sommerville, 2011], this documentation can be considered software in itself. It provides a rich reference framework to understand computer applications and their procedures more comprehensively. These technical documents must be consulted to find collaboration patterns in the software development process and the artifacts supporting it, and to observe the empirical behavior of the free/open metaphor.

- Online forums: The importance of networks is unquestionable, especially the Internet, in the evolution of the modern idea of software. Computer networks are not only a key aspect in modern planning, development, and delivering of computer applications; they are, as stated previously, the rich history of early collaboration metaphors intertwined with the history of software. The early eighties mark a milestone in open collaboration practices in software, with increased code sharing and collective software improvement converged, as a reflection of the four freedoms of free software, and the expansion of computer networks beyond the academic and corporate level. The relevance of Usenet newsgroups in the development of the Linux Kernel by Linus Torvalds is well known [Berry, 2008, 117][Moody, 2002, 42], as well as other

FLOSS projects such as Apache that shared ideas, code, and designs. At that time, almost every FLOSS project had a mailing list to help users discuss plans and make announcements. Fittingly, several journalistic [Moody, 2002] and scholarly [Möller, 2006] accounts on this topic often cite networked artifacts such as Bulletin Board Systems, mail lists, forums, etc. Two mailing lists from the nineties, intended for artists, are noteworthy, "Nettime"[9] and "The Thing"[10][11]. Although their purpose was not software development, these two online communities were undoubtedly forums of feverish discussion on the effects of new media technologies and networks on society. Altogether these information sources could provide skewed angles that can reinforce the central academic-based claims.

- Auto publishing: this is a special kind of publication that is outside the former classifications. These publications can sometimes resemble a magazine, but they are self-published fanzines and manifests where mainly activists, hackers, and enthusiasts dispense their knowledge, expectations, and reviews on the software culture milieu they are living in or anticipating. The significance of these documents is their discussions on open collaboration in computing and the benefits to society of software-based communication that echo the idea of free/open software collaboration metaphor. A classic example is the well-known "Computer Lib/Dreams Machines" [Nelson, 1974]. This self-published manifesto has had, for many years following its publication, a massive influence on theorists and hardware hackers alike, given its advocacy for a computer informed society in a time when computers were still far from everyday life. In the same vein, another publication worth mentioning is the "People's Computer Company" newsletter, which despite its false corporative name, published a series of groundbreaking leaflets in the early seventies, describing the Californian hacker culture. Although these examples reflect a particular mindset that is strongly connected with the well-known American west coast software culture, its influence on later software developments is undeniable.

It is worth noting that most of the existing literature on software studies is in English; this indicates that most of the computer and software companies come from the United States [Campbell-Kelly, 2004], as well as other developments tightly knit to the software saga, such as networks and the computer hardware industry. This challenge is evident in the lack of scholarly

[9] `http://www.nettime.org/`

[10] `http://thing.net/`

[11] These examples show the importance of mails lists and wold not be necessarily used.

works on this topic around the world. Nevertheless, when relevant, literature from other sources can be used. The last chapter will reflect on this issue, providing an examination of diversity issues.

1.6 Literary review

Building a reference system for a research undertaking such as the one this project pursues is challenging. Usually, there is not a single organized corpus covering all the discussed topics in works positioned between disciplines. Moreover, a topic such as software collaboration is difficult to harness theoretically, given the well-known "obscure" nature of software [Chun, 2011, 17]. The ubiquity of computer programs in our lives assembles interests from different and sometimes unarticulated fields. Disciplines such as philosophy, sociology, and psychology are part of the so-called "computational turn"; therefore, they try to approach software as a subject of study, offering a hybrid mindset where interchange is a must. In software studies, one cannot discount the importance of engineering and technical knowledge; it requires a holistic approach to comprehend code and address possible pitfalls of software constructs. The idea here is to mix various methodologies, backgrounds, authors, and disciplines to understand software, collaboration, and impact. The objective is to connect software culture, networks, and collaboration at large while simultaneously supporting the claims necessary to show the influence of software metaphors on society. The following authors and works offer the most useful treatments of these topics.

The contributions of the philosophy of technology provide the general structure of the strong bond between humans and technology. This approach includes several contestations, which are at the very root of the idea of software as unstable media. Some of these accounts refer directly to software, some do not, but as discussed in Chapter Two, the important thing here is to examine society's perception of technology. This is relevant in the sense that –as indicated by software studies efforts– software is perceived as a "dark matter" [Fuller, 2008][Mackenzie, 2006], a statement closely connected to the assessment of technology by western societies. As stated in the introduction of "Technicity": "[Aristotle] institutes hierarchy between theoretical (episteme) and practical thought or knowledge (techne): the Metaphysics and the Nicomachean Ethics, for instance, consistently distinguish between philosophical knowledge —which is an end in itself— and technical and craft knowledge —which is merely a means to an end" [Bradley and Armand, 2006, 2]; this expresses an early idea of technology belonging to the field

of the "un-thought" as opposed to proper "thinking", condemning technology to a secondary and instrumental place. The quote points directly to the prevalent idea of dualism and opposition between philosophy's main constructs (episteme and techne, soul and body, and thoughts and senses, among others), present since the very beginning. This opposition, common in the modern technology narrative, also finds a place within software discussions; it is a tool, an instrument, a prosthesis. Although delving into ancient Greek philosophy is outside of the scope of this project, it contributes to grasping the evolution of this basic idea into modern technology philosophy. Suitably, this study addresses works that are contemporary with the *age of computing* that assess technology as belonging to the external part of the human experience. For instance, in his famous "The Question Concerning Technology", Heidegger critiques the limitations of the instrumental conception of technology; however, in his "essence of technology", he creates an issue of dualism. These unsuitable oppositions must be deconstructed. Indirectly, Derrida brings forth the instability of software and its non-instrumental nature by recognizing that writing should not be subordinated to speech; this suggests the claim that software is not a mere tool. The linguistics operations mentioned by Derrida in "Of Grammatology" ostensibly associate code with writing per se [Derrida, 1997], and they are part of the philosophical claims on software in [Frabetti, 2015]. Also, in the software realm, are the works of Kittler and Hayles. The first author, again, deals with writing, considering it as part of a continuum next to software and follows the idea that technology is not an "extension" (as stated in the classic "There Is No Software" [Kittler, 1997]). Meanwhile, Hayles, in her work based on literary theory applied to computer code, points to software's media materiality [Hayles, 2004].

Appropriately, this burgeoning field of study has provided software studies several updated sources that deal mainly with materiality issues, with the intention of rendering software visible yet unstable. David Berry's work, "Philosophy of Software" [Berry, 2011b], for example, grounds the Philosophy of Technology to the discussion on software, taking into account other trends such as the digital humanities,as a part of the "computational turn", which is also present in [Berry, 2011a]. Also bridging philosophy and software, from a software studies perspective, is the previously mentioned work by Federica Frabetti. Her "Software Theory. A cultural and Philosophical Study", as the title suggest, clearly endeavors to establish a holistic perspective towards the theory of software. Her use of deconstruction and an in-depth reading of Derrida to critically question the idiosyncrasies of software sheds light on the matter. In doing so, she offers a well-constructed understanding of the materiality of software and its nature as a process in

perpetual "un-doing". Her use of a case study based on FLOSS and discussion on software engineering also makes her noteworthy. The book "Cutting Code: Software and Sociality" [Mackenzie, 2006] also makes a substantial claim for the importance of case studies. It presents code studies and software development as a socio-technical process full of pitfalls through a case study featuring an ethnography approach within a telecommunications software project in Australia to complement the idea of software that pervades this project. Likewise, his study on code [Mackenzie, 2003] pinpoints another general aspect of software and technology, its perception as both an opportunity and a threat (another instability). Lastly, the contributions of Lev Manovich are also mentioned. As one of the founders of software studies, he insists upon addressing software as the medium par excellence in his seminal work "The Language of New Media" [Manovich, 2002]. However, it is his "Software Takes Command" [Manovich, 2013] that is particularly important to mention, because, as the title clearly suggests, it considers software a moving force in society. Though his concept of "cultural software" can be confused with the selector of "software culture" chosen here and explained in the next chapter, the clear statements about software and the operations implied in its use, from social perceptions to graphical interface assumptions, offers a wide perspective that places software in a broader discourse.

A historiographical account of the design, development, and distribution of software and collaboration shapes the proposed narrative. Although the linearity of these histories is problematic –as mentioned– most of the so-called "history of" (computation, software, and so on) unfolds around milestones and leaps in the evolution of technology that push the boundaries into the next level. In this sense, the references can be divided into two groups. The first group is general descriptions of computing advances, which uses "stages" to describe the on-going tension between hardware, software, and networks. The second group is focused narratives that revolve around a specific topic, which serve as ever present classifiers to relate the facts to. In the first group, it is worth mentioning the work of Paul Ceruzzi, whose works condensed the history of computing (in America) in a very understandable way [Ceruzzi, 2012][Ceruzzi, 2003]. Although these books can be considered general reference, their organization of facts and tracking of important details have been quoted in several software studies works. The works by Campbell-Kelly, particularly "From Airline Reservations to Sonic the Hedgehog. A History of the Software Industry", closely resembles a *software history*. It offers a historical recount from the business perspective in which software developments can be traced as a consequence of the market forces and its relationship with an innovation-based capitalistic economy, such as the one

usually related to computers. Of particular interest is the account of collaboration and sharing of software as a pragmatic consequence of an increase in the utility of costly hardware when software was not yet a commodity, rather an add-on to the "heavy iron computer" industrial strategies dating back to the fifties [Campbell-Kelly, 2004]. Because FLOSS is a crucial part of the narrative of collaboration in software, scholarly reports of this software are necessary especially to consider the another possible origin –besides business pragmatism– of collaboration in software, the hacker culture. Though significant in several narratives on software collaboration[Berry, 2008][Chopra and Dexter, 2008][Moody, 2002], the hacker culture appears as an incomplete account, too close to a distracting epic. However, the hacker culture is essential in providing a broader context that contrasts the emergence of modern collaborative modes of software production with the more business-related perspective through an alternative view of code, rooted in students, activists, and tinkerers. The work on the open-source movement and the big picture of its cooperation/collaboration (tracing it back to the so-called Unix culture) issues is achieved through works such as [Weber, 2004] and the anthropology efforts devoted to free software by Kelty [Kelty, 2008]. In her book [Coleman, 2012], Gabrielle Coleman explores the ethics behind hacker activity and the process of software developments; she goes on to make several other contributions in [Coleman, 2010] and [Coleman, 2009].

Following the collaboration in software –understood as an entire culture– and its impact on society and networks through the free/open metaphors obviously requires an assessment of how the software development process works and how it is implemented with the support of artifacts and stakeholder interaction. Although it can be seen philosophically, a general discursive framework is drawn from sociology. The accounts of Latour and his well-known Actor-Network Theory inspires considering software development as a socio-technical network (preferring "network" over "system" for argumentative reasons), which becomes even clearer by considering it as a set of relations between humans (developers, users) and machines (in this case, software artifacts used in development). However, the technical part also plays a fundamental role in the narrative. The study of software engineering applied to FLOSS is filled with descriptions like the one from Fogel, one of the early adopters and promoters of the CVS version system, that offers a step by step description of the communications patterns and pieces of software whose intertwined use and coordination produces a piece of software [Fogel, 2017]. Analyzing this geographically distributed and chaotic yet coordinated productive process is Taubert's main concern [Taubert, 2006]. Lastly, in the renowned "Protocol" [Galloway, 2004], the network is described as a

key aspect that lies in the internal gears of distributed FLOSS production. According to Möller, it can be understood by simultaneously considering its cultural implications and the effect of networks' artifacts on the current communications realm [Möller, 2006].

1.7 Document structure

The nature of the subjects discussed in the previous sections already outlines the possible paths to organize the body of the work. The main issue remains to present a clear argument in favor of collaboration as the main force in software development history and as a key component of the software-based open metaphor translated into other fields. Therefore, the idea is, to begin with a more theoretical perspective on software as a tool or as a culture, addressing the archeology of collaboration in software development and networks. Following this, the previously mentioned structure will be addressed by describing the artifacts of modern open software development and how they interact and communicate within the socio-technical network encompassing community and machines. The following is the order of the chapters and a brief description of their main topics and assumptions.

- **Software from a tool to a culture.** To properly assess the effects of software on society and explore the software metaphors based on collaboration that have been translated into other fields, it is necessary first to view software in a broader context. Software must be considered as one of the last expressions within the history of technology. This individual circumstance carries an entire theoretical apparatus in which the philosophy of technology plays an important part. A particularly interesting discussion is the evolution of the very concept of technology and its theoretical journey from being considered a simple tool into a more complex assessment in which society and human nature are entangled with technological developments.The claim here is to consider software as the heir of this relatively new tradition, positioning itself as a neuralgic part of our current self-image and behavior. This step sanctions the claim of software not as a tool but as an entire culture with its own inner logic and where collaboration is evaluated. Concepts such as "technicity" are central to the briefly cited interaction of collaboration and materiality as conceived from software studies.

- **A modular history of collaboration in software.** Although there are several accounts of software history in the literature, the innovation here is the focus on collaboration as the organizing index of the

different applications, systems, and communities in this history. Since the emergence of digital computers as a product of the war effort of the forties, computing devices come a long way to becoming a definitive force in our society. Software, as we know it today, came about at the end of the fifties. However, the concept of programming precedes software, and this is where the opportunity of collaboration occurs. From hardwired programming embedded with hardware[12] to object-oriented programming, software development and deployment is analyzed as a genealogy, more focused on studying the interaction of elements such as programming languages, operating systems, networks, and others related to collaboration, than in offering a linear historical reading of software. Nevertheless, some key milestones in the evolution of software must be taken into account to achieve this goal; the prehistory of software must be considered to include the mainframe era, emergence of networks, hacker culture, personal computer era, and the current Internet era.

- **The open network paradigm.** This chapter will focus on the importance of the relationship between software and networks in the emergence of collaboration in computer program development as practiced today and the specific changes in software philosophies, which have embraced features such as code structuring, sharing, and reutilization as a central aspect of software development. It is no accident that network standards and protocols that have prevailed and evolved into what is now known as the Internet were the ones that embraced the open ethics already practiced in certain software development spheres. However, there are multiple causes to this particularly course of events, which should be addressed. As computer folklore points out, certain cultures coexisting around the computer use stimulated collaborative practices. However, so do very business-oriented aspects such as the late commoditization (and also commodification from a Marxist reading) of software. These tandem software networks will be further explained by examples that show the early cultural implications of the open discourses that already display the emergence of software as a metaphor for other cultural or political undertakings.

- **The nodes of collaborative software development.** Here, software engineering and collaboration artifacts are used to assess how the structure and communication of modern and collaborative software de-

[12]The hardware/software dichotomy poses similar reactions as already discussed in the materiality in software theory; this will be addressed further.

velopment take place. As it has already been mentioned (and will be detailed in Chapter Three), software engineering emerges later than its object of study –software. Since its origins in the late sixties, software has coped with the difficulties of planning, programming, and deploying computer applications, as expressed from the beginning by the very well-known concept of "the software crisis". However, a great effort has been made to come up with novel ways to organize activities concerning the software development process. The focus here will be on two crucial facets: team organization and software tools. In doing, the topics of communication patterns, protocols, and recommendations will be brought forth without losing the framework of human resources management and software tools, such as version systems, issues trackers, and code repositories that are currently being used in development.

- **Conclusions and recommendations.** This chapter will discuss the findings yielded after exploring the proposed hypothesis to answer the research question. It will offer a summary of the narrative addressed to highlight important points to absorb. It will also make a critical reference to some topics that were touched upon but not fully discussed because of space constraints and to prevent distractions from the main objective of this work. Topics such as gender issues and the global vision of FLOSS and software collaboration (focused on the "global south" narratives) will be discussed to mitigate the extended presence of hegemonic accounts, which pervade the software industry. The importance of these hegemonic accounts is unescapable –as attested by the market, individuals, projects, and companies of the recent software history. Their discussion reinforces the premise of the open metaphor from FLOSS and open collaboration. It pinpoints the materiality exhibited by this metaphor by looking at other possible scenarios, in short, showing the required flexibility associated with the very concept of metaphor. Finally, a brief section will be devoted to the already mentioned topic of the maker culture, not as a way of proving a cause-effect relationship of this already complex topic with previous collaborative tropes of software or taking the confusing materiality with the physical substrate of such technologies, but to suggest possible future research directions in which software-inspired open collaboration takes place in other fields; this is almost mandatory under the light of the novelty of the topics discussed, particularly the maker culture. Even though software studies can be considered a discipline still coping with its own identity and principle issues, the alleged tradition of the collaboration in software and its part in the conformation of the open metaphor (of

the software metaphors and generally speaking of the "computational turn") render software somewhat old compared to the personal fabrication, which is seen as "the next big thing". However precisely herein lies the importance of taking this chapter one step further into the discussions of possible new research paths to try to explain the called maker culture from a critical perspective. The dangers of such labels such as "revolution" are well known and used frequently by mass media; it is no surprise that such adjectives were also applied to FLOSS in the nineties. While there is no doubt about the potentially disruptive nature of collaborations patterns in personal fabrication, more instruction is needed for academic approaches to the maker culture; this is an important issue that will be discussed in this chapter.

2 — Software: from a tool to a culture

2.1 Introduction

The first premise to engage in a fruitful discussion on software and its implications is to recognize software as a technology. Although this can be easily assumed and does not require debate, in doing so, several other aspects of the condition of software in our society are revealed. Therefore, to fully understand the social implications of software, we must focus on grasping the entire concept of technology, its contradictions, its assessments in the western thought, and its recent leap into the theoretical spotlight. The objective is to draw from a broader issue, such as the current perception of technology and its evolution. This process will reveal contending factors that can be handed down to the software realm. However, software is not just another instance of the technology construct; it is an artifact belonging to an honorable list of examples subject to debate. In short, software is "the engine of contemporary societies" [Manovich, 2013, 6] and, as briefly hinted in the previous chapter, exhibits an unstable nature. Although technology may share similar conditions, software has particularities that distinguish it from other technologies. Two commonly discussed key aspects concerning these topics, instrumentality and materiality, illustrate both possibilities; that is, what software shares with technology and what belongs to its discursive universe. The first one has a long tradition in the history of technological thought; it is currently a subject of regular dispute, as it places technology at the center of the human condition, not attached as a tool. The second one is more difficult to assume, and its approach involves abstractions and complex arguments to prove a mistaken intuitive idea on technology perceiving it as obscured and not easily visible. Thus, this examination will present software as a machine, but an exceptional one, a machine whose nature is very entwined with the concept of computer and is perceived as a universal

machine. This chapter is written combining ideas from several disciplines, usually referenced to explain software (apart from the obvious computer science). The affirmations and observations unfolded in this approach support the claim that collaboration is not foreign to the ways of software. That said, this chapter will trace the journey of software from a tool to a "culture", as well as the conditions that are relevant to articulate a philosophical, even metaphysical, condition for collaboration in software. This condition is the foundation to view collaboration from the other two main argumentation points (historically and pragmatically). Though the discussion on software remains mainly abstract, when possible, concrete assumptions and partial conclusions directly connected with collaboration will be made

2.2 The problem of thought in technology

There are several accounts of the ancillary place given to technology in the western philosophical tradition. Hence, the complete turn that the notion of technology has made in the last 150 years is surprising. It has acquired a dominant place in the current theoretical inquiries of the most diverse disciplines. A brief overview of the origins and recent evolution of this ongoing discussion will set the framework to assess software and technology at large. I use two sources that reflect two diverging approaches to the issue.

First, we have to consider ancient Greece. As stated, in reading Armand and Aristotle's philosophy, there was an original process of obscuring technology. The previous can be placed in the foundational and platonic dichotomy between *Episteme* (Knowledge) and *Techne* (Art), used to describe abstract processes of thinking in the first case and manual labor in the second. This binary construct defines knowledge as an exclusive property of Episteme, relegating Techne to a position that involves no system of thought; thus, there is no knowledge in techne. A corollary usually inferred from this dichotomy is the instrumental nature of technique and technology, which are seen as a means to an end.[1] However, this fundamental dichotomy is not as apparent as one might think. Consider, for example, Aristotle's definition of "Art", offered by Layton and taken from an introduction to Aristotle by Richard McKeon: "[n]ow since architecture is an art and is essentially a reasoned

[1] Another issue arises here, and it is the difference between technique and technology. Some scholars see no difference; others see a complex relationship, with technology as a complete system and technique as just a part. Although I concur with the latter, I consider it essential in the discussion of software as a culture. A proper explanation and position will be attempted later.

state of capacity to make, and there is neither any art that is not such a state nor any such state that is not an art, art is identical with a state of capacity to make, involving a true course of reasoning" [Edwin T. Layton, 1974, 33]. Two things are inferred from this quote. The first is that art is the "capacity to make", namely, a direct relationship with technology, as the techne is usually understood. The second is "art" involving reasoning. The second conclusion directly contradicts the assumption of technology as "without knowledge". The argument developed by Layton using this quote is compelling. It places Aristotle as the likely founder of philosophical inquiry on technology, and –although his use of Aristotle diverges from Bradley and Armand's [Bradley, 2006]– his approach reveals some common points in the critique of technology. In turn, his assessments become an object of further critique. As a first step, let us evaluate some of his primary considerations. He is concerned, as Bradley and Armand, with the subordination of technology to science. Layton notes that the broadly accepted model of technology as an applied science is firmly rooted in a particular formulation (drawn from "A History of Technology" by Charles Singer, E. J. Holmyard, and A. R. Hall). This formulation states that from the century 19th onwards, technology acquired scientific content. This thesis reinforces the idea of science as the producer of discoveries, and technology (misinterpreted as the equivalent of technique), a mere applier of these new findings, borrowing knowledge from an external source; this is also known as the theory of relationships between science and technology. Building upon the critique of this extended thought, Layton proceeds to denounce a contradiction in this denial of thought on technology: "[t]wo assumptions are critical here. The first is that technological knowledge is essentially identical with natural philosophy. The second is that this knowledge has been produced by scientists since 1800. Logical deduction from these premises leads to an absurdity: that prior to 1800, technology involved no knowledge at all" [Edwin T. Layton, 1974, 31]. We could further address Layton's reflections on this issue, but there are too many authors and concepts involved that we would exceed this work's objective. However, there are three important reasoning against the secondary character of technology and its perception as applied science. First, we must mention a very typical –even romantic, by some accounts– historical movement, the employment of the figure of the craftsman.

The pursuit of showing that indeed there is knowledge in "doing" and technology inevitably requires exploring pre-industrial revolution times (before the alleged "scientification of technology" in the 19th century), tracking back to the middle ages and craftsmanship. The ability to manufacture goods with one's own hands displays an evident *know-how*, hence, *knowledge*. Layton

goes as far as pointing to Marxist readings on the industrial revolution that show an intention of equalizing science and craftsmanship. He states, referring to Ziel: "[h]e held that the scientist was a hybrid combining the craftsman's empiricism and the scholar's systematic thought" [Edwin T. Layton, 1974, 35]. Although this statement can be considered an exaggeration, it points in the right direction. Science and technology are not isolated; they are interrelated but not in a hierarchical fashion. Layton also promotes the idea of *community*; this is inferred throughout the article quoted below. The community is vital for two main reasons. The first is that it allows for a social and historical commentary on the history of technology, let alone that the community is affected by technology. The second reason is that it establishes communities as subjects of communication exchange that aims to present science and technology as peer communities. This, in turn, reinforces the historical perspective via the evolution that the definitions of "knowing" and "doing" have among the communities. The third reasoning is the drawing of a historical continuum from the artisan to the engineer, with the latter embracing *design* as the embodiment of an iterative process, a problematic "reasoned ability to make"[2].

One of Layton's main assertions, which summarizes his whole argument, is that Technology is a system of thought on its own, different from science but with its set of rules and knowledge. The role of the engineer in the building of the society (community) is also of paramount importance, as one can infer from the title "The Revolt of the Engineers: Social Responsibility and the American Engineering Profession", one of his most recognized works. Fores[Fores, 1979], makes an exhaustive analysis and critique of these works by Layton. He (as Layton) advocates for the emancipation of technology from science. He also concurs on the engineer, as it relates to the craftsman tradition. He states, "[t]he use of scientific knowledge and practical skills were both essential preconditions for achievement; and no one can sensibly rank one against the other" and "[i]t is clear that the engineer as an 'artisan' has not been replaced by anyone sensibly thought of as a "scientific practitioner". Nor will he ever be; for engineering, which is an art- making use of scientific knowledge (just as any other art does), is not that sort of process" [Fores, 1979, 859]. This "making use", in turn, brings the evident pragma-

[2]This discussion, though a little abstract, could be translated easily onto collaboration in the software realm, reflecting indeed, similar problems. The community is a universal construct among the FLOSS discourse, referring users but especially programmers. Design and Engineering are also constructs that can be disputed when one considers the ongoing critique of software engineering. Here again, the tension between design and hands-on trial and error approaches to software development is regularly addressed.

tism that impregnates Fores's entire argumentation to the foreground. Let us focus on two parts of this pragmatic position. First, for Fores, much of the discussion lies in language inaccuracies, as he points out in his observation of the differences between English and German. Making no difference between Technology and Technique, he asserts, referring to Germans, that "in an uncomplicated way, they are happy for manufacture simply to be a part of Technik, a third culture separate from Wissenschaft (science) and Kunst (art)" [Fores, 1979, 852].

Similarly, Fores seems to consider the processes of deep reflection on Technology unnecessary. The evidence of the importance of technology within social history should be evident from a merely instrumental perspective. He states, "[t]he best way, surely, to connect the history of the manufacture of useful artifacts with social history is not to stress intellectual questions at all but to point to the widespread use and utility of these artifacts" [Fores, 1979, 857]. The main argument that summarizes Fores's pragmatic position is to point out the so-called "rational fallacy", which can be briefly described as the justification of knowledge in technology as a result of an influence of science, either through a subordinate construction of "applied science", a strong science-technology relation mechanism in which some steps in manufacturing are based on science or by assuming that the technology professional is a "scientific technologist"[3].

The discussion of these two quite divergent points of view clarifies some critical working assumptions, that is, premises that I consider necessary to continue this work. I believe Layton is right to see technology as a system; however, he relies too much on science and design as a tool for translating scientific findings into product production through the figure of the engineer as a scientific technologist. Similarly, Fores correctly points out the extreme practice of technology not related to science. However, his exaggerated pragmatism leaves aside other theoretical developments, something that is quite contradictory, considering that this is a paper on technology. Fittingly, the first assumption is how technology and technique relate to each other. Fores's

[3]Paul Graham made similar comments in the field of software development. He pointed out the drawbacks of the term "computer science" and how practical disciplines somehow feel the need to be more mathematical to make their findings more respectable. He states, "[i]f you find yourself in the computer science department, there is a natural temptation to believe, for example, that hacking is the applied version of what theoretical computer science is the theory of" [Graham, 2010, 21]. This wording resonates directly with the critique of applied sciences and directly relates to hacking, understood as computer software programming.

social concern that technologists and technicians should not be considered different (because of the negative social and economic impact on the latter) is understandable. However, while such division of labor may be criticized from a political point of view, there are more theoretical advantages in following the mentioned distinction. Hence, "logos" in technology make it different from (and more complex than) the technique.

Moreover, the recurring discussion of viewing technology as a mere tool can be solved by the following construction: technique as a way of addressing the instrumental aspect of technology. In turn, technology is a system of thought, independent from science, with its own rules. This relationship is one of inclusion; technique is included in technology's system of thought. It gives technology enough space to maneuver to make possible abstract operations such as the isomorphism expressed by the concept of the metaphor, as seen in open collaboration. It also allows us to consider a particular technology, such as software, as a complete culture. The second assumption emphasizes the practical nature of technology, a nature that is itself reflective, but not comparable to science and closely associated with the craft of the maker. Engineering also benefits from this perspective, but as it will be explained later, it still carries problematic interpretations, especially when speaking of software engineering and its understanding of the software development process. These two assumptions (that can also be seen as pragmatic) aim to shift the focus from a pragmatism-rationalism dispute to a more productive discussion.

2.3 Arriving at the turn: some steps towards modern critique on technology

As stated in the previous section, there is an ongoing critique of the very idea of technology, beginning with the handling of idealistic dual forms. The account offered presents two points of view that put the major issues of technology currently in discussion into perspective. As suggested, a turning point in the instrumental narrative of technology comes with the industrial revolution. For example, despite their differences, Layton and Fores's texts revolve around the assumption that this change (industrial revolution) took place in the 19th century (although when it started precisely and its influence on science from this century onward remains a source of controversy). This reflection on technology is the starting point to discuss some ideas of the leading theoreticians who addressed the problem, mainly in the 20th cen-

tury. Although a linear historical approach is far from this endeavor, it is remarkable how the ideas influenced each other by weaving threads between different positions. With that in mind, some of these ideas are briefly discussed and interpreted to obtain a greater understanding of technology.

One of the first thinkers to consider technology from a broad perspective and care about the outcome of the industrial revolution was Marx. His commentary on the issue, although too extensive to be covered in detail, has two interesting questions useful for researching technology and software in the long term. His ideas still have a significant influence on the efforts of social theories, particularly the so-called critical theory. Specifically (and perhaps more important for the argument), he makes a critical conceptualization of technology, where he includes instrumental concerns, social and economic observations, and the ontological nature of technology and the machine. Although the partnership worker-machine appears as an instrument of profit within capitalism, the relationship is not direct. If in order to consider technology a mere instrument we must consider it as an external part of ourselves, an *extension* (explaining its "without knowledge" nature as opposed to the immanent knowledge of the human condition), in Marx, the relationship appears as a complex network of mutual interchanges between human and technology (machines) that resembles more a "system"[4]. The result then is a technological conception that overcomes instrumentality and postulates technology as a force that shapes society (and vice versa). According to Shaw, "[f]or Marx, technology is not a neutral force that only becomes an instrument of exploitation in the hands of the ruthless capitalist (...) As Marx demonstrates these changes are also responsive to, and implicated in the development of, social institutions like the law and the family" [Shaw, 2008, 12].

On the other hand, the mentioned social and economic factors, including popular knowledge, inform Marx's work when discussing the role of technology in increasing human capacities that generate a surplus that benefits capitalism. The well-known resulting construct of "alienated labor" and the very nature of capitalism results in an ontology where "competition" and not "collaboration" is at the center of the human condition, almost a biological impulse strongly linked to subsistence. As Benita Shaw mentioned, this premise can and has been widely discussed. The underlying concept of the influence of technology on ourselves is consistent with reality. She states,

[4]This statement is the key to the whole argument. If people and technological devices belong to a broader framework, a system that ensures communication between actors, collaboration is a possibility. Software is a particular and very special instance that facilitates this process.

"[t]he study of Technoculture, therefore, must necessarily engage with the sense in which the technologies which are an inseparable part of our social worlds, also produce changes in how we conceive of ourselves" [Shaw, 2008, 14]. Alienated labor and this ontological reasoning are coalescing conditions of Marx's dialectic of technology and directly affect the existing concept of the "community". As discussed, a community can be seen as a social body affected by technology or as a set of conferring communities that influence each other, such as in the case of the technology-science duple. For Marx, under capitalism, the community constitutes a sort of battlefield for the worker: "[t]he worker is further alienated by being in a relationship of competition, rather than cooperation, with other members of the community" [Shaw, 2008, 13]. Despite the negative connotations expressed in this view, where technology could be taken as a means to an end and worse, include humans as part of a tandem with the machine in an exploited system, the Marxist assessment of technology emphasizes the integration of man and machine as an element of the larger framework (or system) that contains it. This humanist technological reasoning then is not direct but supports and underpins this ontological operation. As Louis Armand stated, "[f]or Marx, the machine no longer represents a counter or post-humanistic development, but rather the very 'essence' of humanism" [Armand, 2006, 45]. Summarizing Marx's complete vision of technology would be impossible. However, another feature of his work not yet mentioned, besides the ontological-humanistic reading presented, would be his historical method to study the always-changing relationship between man and machine (machine as an instance of technology) always through the prism of economic concerns.

As observed by Stiegler, the work of Heidegger is seemingly the modern entry of the philosophical analysis into the field of technology, given that "*modern* age is essentially that of modern *technics*"[5] (emphasis in translation) [Stiegler, 1998, 7].In his approach to the issue of time, an ontological substrate and the introduction of "ratio" namely, calculation related to reason, are some of the discursive motifs that resonate with the idea of technology intertwined with human nature (ontological perspective) and with computers (software in general) as a consequence of the conversion of reason into calculation. On the other hand, the discussion on instrumentality remains, as it is explicitly expressed in the well-known "The Question Concerning Technology": "[i]nstrumentality is considered to be the fundamental characteristic

[5]Because the original is in French, a similar idiomatic discussion could be held on the term "technics" as the already mentioned in the case of German. Although the statement is whether one selects technics or technology as the main intention, for the purpose of this paragraph, technology would be preferred.

of technology. If we inquire, step by step, into what technology, represented as means, actually is, then we shall arrive at revealing. The possibility of all productive manufacturing lies in revealing"[Heidegger, 1977, 12]. This phrasing does not leave aside instrumentality; instead, it states that it is something more, a "revealing". His movement is, at some point, compatible with the proposed pragmatic organization of technique as a concept that covers the undeniable instrumental concern of technology and technology itself, as a system of thought that contains technique but is complex in nature.

It should be noted that "revealing" is understood as a form of truth inherent in the development of technology, where "manufacturing" allows us to refer to the above discussion on making and craftsmanship. This "revealing" is the key to understand this seminal Heidegger text and its widely quoted and productive quest to address the hidden "essence" of technology. The assumption of technology as an applied science is also misconstrued to reconcile a duality between explanations of technology from an instrumental perspective or an anthropological point of view. Therefore, to the extent that this well-known essay covers the discussion on such important constructs, it is of recognizable preponderance. Moreover, the two pillars of the entire argumentation contribute more specifically to a conceptualization of technology with a discursive space for collaboration. These so-called "pillars" are represented in the concepts of "standing reserve" and "enframing". The standing reserve can be assumed as a descriptor of the potentiality that lies within the material agency of technology and its ability to perform (a particular task or a set of tasks).

Nonetheless, the former consideration could again lead to instrumentality when humans are not thought as part of this standing reserve. When this potential is considered as a mere resource, the dichotomies discussed, such as *human/nature*, move forward, recalling an ontological motivation to consider the human being as part of the "standing reserve", in an attempt to solve this inconvenience. On the other hand, and as an example of the complexities of the human-technology relationship, if we consider humans only as part of the standing reserve the risk of instrumentality falls on them. Although the forces of other systems (such as capitalism) can exert pressure on the individual, it is he or she who "orders" the resources through "ordering"[6].

[6]The English translation used in the text, although semantically ambiguous, is revealing concerning the meaning of "order" and the related noun, "ordering". On the one hand, in technology, it is the command to give an order to something. On the other, it is the logical operation of arranging elements within a system. Despite their seeming antagonism, in reality, these two possibilities allow us to address different aspects of the problem.

A possible follow-up to this notation on ordering could be the need for calculation. This observation goes back to rationalism and, as will be briefly explained, has several implications when considered from the perspective of computer science. The other pillar of Heidegger's argument in his search for the essence of technology is "Enframing". He states, "[e]nframing means the gathering together of that setting-upon which sets upon man, i.e., challenges him forth, to reveal the real, in the mode of ordering, as standing-reserve. Enframing means that way of revealing which holds sway in the essence of modern technology and which is itself nothing technological. On the other hand, all those things that are so familiar to us and are standard parts of an assembly, such as rods, pistons, and chassis, belong to the technological. The assembly itself, however, together with the aforementioned stockparts, falls within the sphere of technological activity; and this activity always merely responds to the challenge of Enframing, but it never comprises Enframing itself or brings it about" [Heidegger, 1977, 20,21].

A less profound but suitable interpretation of this passage, places "enframing" on a higher-order in Heidegger's narrative, which incorporates the ontological logic of man being "challenged forth" and the mentioned "standing reserve" as ordered matter. The issue of materiality is briefly related by directly addressing "revealing the real". Moreover, the central theme of modern technology appears to transcend previous conceptions, leaving room for hybrid assemblages. In my opinion, this is the most critical construct that can be extracted from "enframing". As its translation suggests, it offers a framework or assemblage where people and technological elements can relate in an orderly way, giving technology potentiality or "standing reserve". Instrumental concerns are dismissed given that the assembly is located *in* the enframing and not the other way around. That is, enframing appears as a concept that encompasses others and provides a sound environment to reveal the essence of modern technology, which by the way, and using Heidegger's phrasing, "in itself nothing technological". This last statement recognizes the hybrid nature of human-technology negotiations and again hints at an ontological component. These ideas have often been mentioned when discussing modern technology, which obviously arises from some critique. For instance, Armand points out that this essence recalls the dualism commonly associated with idealistic positions, establishing a distance with the intended materiality. From another perspective and examining the question of technology, there are some controversial statements, which reshape the "rational fallacy" mentioned in the previous section. For example, when referring to the relationship between technology and science, the text states that "mathematical physics arose at least two centuries before technology" [Heidegger,

1977, 21] and that "[s]urely technology got under way only when it could be supported by exact physical science" [Heidegger, 1977, 21,22][7]. Despite the difficulty associated with Heidegger's texts and that, in this case, no mention was made of computer technologies (let alone software), a link can be made by assessing the following passage: "[i]f modern physics must resign itself ever increasingly to the fact that its realm of representation remains inscrutable and incapable of being visualized, this resignation is not dictated by any committee of researchers. It is challenged forth by the rule of Enframing, which demands that nature be orderable as standing-reserve. Hence physics, in all its retreating from the representation turned only toward objects that has alone been standard till recently, will never be able to renounce this one thing: that nature reports itself in some way or other that is identifiable through calculation and that it remains orderable as a system of information" [Heidegger, 1977, 22]. Several points can be inferred from this passage. Science (physics) deals with representation problems that are difficult to assess. Enframing and the standing reserve configure nature as an orderable system; this implies calculation. This system is, in turn, an "information system" which opens a discursive space for technology (machines) to process this information. From Heidegger's reading and contribution to the main thesis, we can conclude that although the whole could be seen as an argument favoring technological determinism, the problematic exchange between hybrid human-machine assemblies, the lack of a "committee of researchers" to decide on the technology-science relationship, and the unattainability of a process to reveal the essence of technology makes the whole apparatus unstable and difficult to manage.

The concept of "human technicity" will be the closing stage in this general approach to some of the modern (after the industrial revolution) discussions and critiques on technology. Unlike the previous concepts that centered on the contributions of two particular authors, the concept of human technicity looks to others, mainly, Clark –who coined the term, Derrida, and Stiegler. However, I draw more from the latter because his work establishes a dialog with the others. The importance of Stiegler's reading of human technicity lies primarily, for this work, in the following three crucial aspects: the intimacy of the relationship between humans and technology, language, and especially writing. This form of technology is also deeply entangled with the human condition and history as an inevitable process of understanding

[7]Although an additional explanation is provided to offset the statement "(...) this is correct. Thought historically, it does not hit upon the truth" [Heidegger, 1977, 22]. The entire concept remains difficult to agree upon.

technology. Let us begin with some remarks from Stiegler. In his well-know "Technics and Time", a project with high theoretical aspiration, though too extensive to be summarized, he traces some questions of modern philosophy on technology. In a critical light, Stiegler highlights Marx as a precursor and Heidegger as a key player in those efforts. He states, "[t]he meaning of modern technics is ambiguous in Heidegger's work. It appears simultaneously as the ultimate obstacle to and as the ultimate possibility of thought" [Stiegler, 1998, 7]. This phrasing is reminiscent of the widespread conception of technology as danger and opportunity; however, it also points to technology as "thought", which subtly deals with the obscurity of technology and its well-known dichotomy. As noted, Stiegler agrees with Heidegger on the need to question technology, but the essence that lies between this opaque matter –technology– remains a concept external to our condition (we do not know what it is). Here, the problematic concept of an "ideal truth" which is somehow concealed, emerges again. Instead of following this path, Stiegler and, in general, the theoretical apparatus of "originary technicity" offers a kind of essential explanation that is, however, more entangled with very materialist concerns such as language, history, and our human condition. The problem then shifts from "essence", which is not technological, to a premise where the relationship between humans and technology has always existed, and where we really are "technological". "Stiegler has developed a highly original philosophy of technology, the central premise of which is that the human has always been technological" [Hansen, 2004]. Moreover, supporting Derrida's deconstruction theory, this movement partly enables the dichotomy between humans and technology to be overcome and the issue to be addressed with a more robust approach. To briefly explain this apparatus, let us consider technology and its objects (tools); they are a record of our history (for example, an ancient piece of pottery), therefore, they are the object of archeological inquiry[8]. However, not all technology is intended to record except the ones called *mnemotechnics*.

Writing is a form of technology and belongs to the latter category (mnemotechnics). In ancient Greece, Plato noted its quality as a technique to aid memory. The implications of this observation reinforce the discussion of *episteme* and *techne* in technology. If western philosophy considers technology obscure and "without knowledge" the same can be said about writing. Plato viewed writing's aiding capacities with contempt because it was used by the Sophists

[8]Stiegler references the work of the French paleontologist André Leroi-Gourhan "who tightly connects the appearance of the human with tool use" [Frabetti, 2011, 6]. Although Stiegler also sees a connection between the emergence of humans and tools, this path was not explored further for reasons of space limitations.

who were perceived as more interested in the form of an argument (hence their use of writing to *remember*) than the truth. Thus, we are left with technology in general and writing in particular as removed from knowledge and both instrumentalized. If technology has been widely assumed as a mere tool and writing, in turn, is frequently thought of as subordinate to speech; this signals that "the devaluation of technology in western philosophy goes hand in hand with the devaluation of writing" [Frabetti, 2011, 6].

Furthermore, if writing and technology, in Derrida's terms, support the inscription of memory, a logical follow-up is that they are connected to the past, and therefore history. The inclusion of history is of overriding importance because it gives Stiegler a binding construction to reinforce the position of "originary technicity". It could be argued that we obtain knowledge about ourselves as species through history and through writing, the mind executes and operates that hard exterior; it help us to understand ourselves, as is often the case when one tries to explain something through written words. In short, we have this originary technicity, or better yet, as Stiegler calls it, "originary prostheticity". It indicates the human need for external objects to survive, encompassing technology and writing, in the process, hinting at a "technical memory" and a temporal element that is indispensable to understanding our condition.

It should be noted that this brief account leaves many other important authors and concepts out of the discussion. However, I decided to focus on those selected to straightforwardly show how the baseline of "originary technicity" addresses the issue of technology in western thought and the relationship of concepts such as community, technology (included the machine), and language (writing). The interaction between these factors is fundamental when trying to understand software firstly as a technology in general and a symbolic machine specifically; secondly, software as writing, as a consequence of its symbolic nature. When viewed broadly, the potentiality that software offers the community again surpasses its use as a tool; this is represented in collaboration. In turn, this possibility of collaboration, which is very material and real, as several software examples have shown, demands a particular arrangement of actors (human and machine) where writing in the broad sense plays a key role. Accepting these premises makes software also a medium, which again reflects the close link between humans and technology. This argument will be developed throughout the rest of this chapter, and, where necessary, additional information will be drawn from other theoretical accounts related to originary technicity.

2.4 The Computer: a universal and flexible machine

Until now, the discussion on technology has been mainly abstract because of the emphasis on the philosophical approach. However, there is a clear intention of discovering some elements that are transversal to all types of technology, hence, software. As discussed, these elements also engage in a dialog with the human condition in which there is a relationship of mutual exchange between humans and technology. This relationship can be considered ontological because of its importance in the constitution of our technological character. In short, humans shape technology, and technology shapes humans. However, to advance the discussion, it is necessary to consider some of the particularities of specific technologies related to computer applications. Technology is revealed through this step, confirming the challenges of its assessment by using the particularities of software, in general, as an example of these challenges. Software has particular attributes that are at its core that portray it as an opaque and even –mistakenly– immaterial technology. However, to get closer to the nature of software, I first use two intertwined concepts, the machine and the computer (computing).

In previous sections, when talking about technology, I have very freely referred to *the machine* as a theoretical construct that comprises some aspects related to technology. However, this does not mean that the machine is equal to technology at a discursive level, rather, that it can be seen as a material manifestation of it in our lives. So, the obvious question that arises from this position is: What is a machine? As in the case of western thought's perception of technology, this question is difficult to answer, let alone considering space limitations. Moreover, I am convinced that defining the machine is an arbitrary exercise, which can become futile if a monolithic, non-contradictory answer is expected. However, in trying to do so, the qualities necessary for my claims appear. Let us begin by considering the machine like a system, a set of different parts that interact with each other (with a specific task in mind) through inputs and outputs. This first step brings up other considerations that must be addressed.

On the one hand, there is the question of whether all the parts should be mechanical, even better, artificial (created by humans). On the other is the question of whether a one-piece tool can be considered a machine. These two questions are intertwined. Consider a stick used as leverage, one could ask, is it a machine? If machines have several parts and we only look at the

stick, then the answer is *no*. However, if we look at it as a system involving the lever, a weight to be lifted, and a person applying force, the straight negation becomes weaker, and the answer more apparent. These types of examples are widespread and somehow indicate that the idea is not to shift the problem from *machine* to *system* but to try to negotiate various aspects to build a working base. To be assertive and pragmatic, let us suppose that machines are composed of several parts, sometimes inorganic and sometimes fused with humans, depending on the perspective and the focus, but mainly as a condition when considered as a system or as a part of a unit; consider modern software[9]. Software could be seen as a *symbolic machine*; the intricate structure of modern software confirms that it has a complex character. Parts of the software could be seen as artificial, representing a machine with inputs and outputs in a very traditional way.

However, the boundaries become blurred when one considers that the interactivity and feedback require constant communication with a user to provide this machine with readiness and purpose. Of course, this particular configuration resembles cybernetic principles where the relationship human-machine is symbiotic. As Stiegler points out regarding Heidegger's thinking on technology, "modern technics is dominated by cybernetics as the science of organization" [Stiegler, 1998, 23]. This quote reveals to us what is essential in a system composed of machines and humans, organization. When dealing with the duality of inorganic-organic beings, Stiegler mentions the need for a third category in the form of "inorganic organized beings" namely, "technical objects" [Stiegler, 1998, 17] [10]. This construct not only brings us back to the mentioned assessment of technology as a "system of information", but it also stresses organization, structure, and, more importantly, the associated need for calculation, which partially justifies the emergence of the computer as a machine to calculate. In short, a systemic view offers a framework where symbiotic relationships between humans and machines take place in a complex way [11]. This arrangement, in turn, generates an environment

[9]Here, the assumption of *modern* is preponderant because primitive software was not Interactive; this brings up several considerations when approaching software from a philosophical and historical perspective.

[10]It is worth noting that this development acknowledges the impossibility of reducing technology exclusively to either a biological, mechanical, or anthropological explanation. Here, the opportunity to undertake a hybrid approach can be considered a logical step.

[11]Here, it is worth mentioning Bruno Latour's concept of "actant", which is part of the so-called Actor-Network-Theory. The concept of actant involves machines and humans and though both are different, they are capable of having agency. Apart from disagreements on the issue, the idea of an equalizing agency between humans and machines through a network (a computer network) is of interest and will be revisited in more detail in Chapters

in which "ordering" and "calculating" are operations that technologies (and machines), such as the computer, can help to perform. However, considering the computer exclusively as a calculation machine is not only imprecise but, historically speaking, would also suppose an instrumental reading of technology as subordinated to science.

The computer as a technology and a machine is relevant because it offers a discursive field that takes place just before the appearance of software, and for obvious reasons, remains closely connected to it. I have already stated that software is a symbolic machine of sorts, and it can be said that its appearance (like the concept of stored program) is actually indebted to the improvement of the computer as a calculating machine. Therefore, to finally establish software as a machine, I will briefly discuss the computer. In doing so, the relationship between science, technology, and even engineering is once again brought into question. As mentioned, computing requirements (related to science) are at the origin of the computer machine, but they are only a part of the effort towards this technological development. In fact, the power and particularity of the computer as a machine for calculation is indebted to the different disciplines whose interrelationship is decisive in its emergence in the period from the late 1930s to the mid-1940s of the 20th century[12]. According to Trogemann, *"[d]ie programmierbare Universalmaschine wurde möglich durch die Zusammenführung dreier weitestgehend unabhängig verfolgter Forschungs – und Entwicklungslinien: der Formalisierung in der Mathematik, der Mechanisierung durch das Ingenieurwesen und der rationalistischen Tradition in der Philosophie und den modernen Wissenschaften generell"* [Trogemann, 2005, 115]. Let us briefly discuss these three mutually related parts of the computer thought.

Firstly, the formalization of mathematics is strongly connected to language; mathematics expresses language using written symbols. Although in the previously discussed theoretical construct of originary technicity, language (writing) is considered a technology that supports the externalization of memory, in this case, this writing does not respond directly to an externalization using natural language but to an abstraction that responds to the demands of calculation. Thus, technology systems can be associated with an orderable field using calculation and symbols with an issue of correspondence. This last statement demands special attention; it somehow refers to a premise

4 and 5.

[12]It is common in the history of computing to refer to several precursors that date back to Babbage, Leibniz or even earlier. However, because my interest is in software, the emphasis will be on the particular period when a viable version of software became visible.

behind notation and symbolic machines, namely, that external phenomena can be translated into symbols, processed within the symbolic realm, and the results are translated back to the original problem. Turning back to mathematics, its development has been brimming with enhancements that are key to the development of the computer, from the contributions of the notable Al-Khwarizmi to developments in calculus and the introduction of Boolean algebra, among others. However, one advancement must be singled out, the Turing machine. Proposed as an answer to an exclusively mathematical problem (the *Entscheidungsproblem*), it is essential because the theoretical framework behind it paved the way for future symbolic developments on which a programmable machine could be developed. Although the Turing machine remained a theoretical experiment and its parts, impractical (such as its infinite tape), the combination of symbols, statuses, and precise mechanism of operation represent a milestone. Symbols were input to instruct the machine, and the results were output also in the form of symbols. The Turing machine was also partly responsible for the theory of automata, which focuses on the importance of the computer as an automaton. This association of the computer (primitive computer) remains essential because "the stuff of computing is symbol structures. Any automaton capable of processing symbol structures is a computer" [Dasgupta, 2016, 12]. The prospect of a machine of symbols based on one instance of a language opened a whole world of opportunities that would soon exceed the realm of calculation.

However, the computer was also the product of engineering efforts. This statement brings us back to the discussion on the tension between science and technology and the role of engineering. As stated in the second section of this chapter, the relationship between science and technology arouses debate with engineering in the middle. In this narrative, technology aligns more so with practical endeavors that are not directed by scientific formulations but are based on a trial and error approach with a strong tinkering component. Having said this, especially concerning computers and despite the mathematical precedents driving the computer as a symbolic-calculating machine, there have also been practical developments that have not always responded to this symbolic evolution. For instance, the use of a punched material as a medium to instruct the machine. Notably used by Konrad Zuse in his first computer, the Z1 (1935- 1936) –allegedly the first programmable computer– it dates back to Hollerith's tabulating machines (late 19th century), Babbage and his Analytical Engine (1837), and the Jacquard loom (early 19th century). Although the first three are closely related to calculation systems, the Jacquard loom was directed to a fabrication process (textiles) and the pursuit of simplification. The industrial revolution required the essence of

the machine to be more pragmatic, and use punched cards to automatize the production of brocade patterns. Here, it is important to rethink the machine as a process of grammatization, following Derrida's description of a supplement as a way of discretizing a flux [Stiegler, 2006]. The Jacquard loom, then, could be seen as a mechanical externalization of the textile workers' "muscular" actions materialized in a machine.

Similarly, the electronic-based computer was born in the 1940s as a device designed to aid or enhance the computing tasks performed by humans, who were, in fact, the "computers" at the time. Their calculation duties where progressively shifted to the machine: "[...]in the 1930s and 1940s, people who were employed to do calculations –and it was predominantly women who performed this clerical labor– were called 'computers' [. . .] the different interpretations of the sentence from World War II to the end of the twentieth century mark a shift from a society in which the intelligence required for calculations was primarily associated with humans to the increasing delegation of these labors to computational machines" [Hayles, 2005, 1][13]. Thus, to close the discussion, the computer is also a product of applying engineering to a calculation problem. More than muscle labor, the computer outsources human computers' intellectual work and their system of mechanical calculators and mathematical notations. Perhaps the convergence of the computer's pragmatic and theoretical path was materialized thanks to the von Neumann architecture, ostensibly the first computer using the stored program concept –a direct precursor of software– and other advances. Because of its configuration, programming was a consequence of computing, "[...]the von Neumann architecture separated the physical calculating parts from the tasks. By storing commands in the electronic memory, it began to free the computer engineers from the physical constraints of the system. A new class of engineers arose: programmers" [Brate, 2002, 67].

The third element to analyze is the rationalist tradition in scientific thought. As a product of the scientific and philosophical movements of the 16th and 17th centuries, the universe was seen as a book whose static laws were available for reading by following organized steps and methodical inquiring in a sort of algorithmic thinking. Under the precepts of this paradigm, observation is required to understand natural phenomena through measurement. With the support of the already flourishing mathematical notation, the promise of predictability became conceivable. This means that exper-

[13]Hayles refers to a passage from the novel "Cryptonomicon" by Neal Stephenson, which she used to illustrate the mentioned use of the word "computer".

imentation, combined with symbolic operations through scientific measurement, led to the idea of computing. Thus, the hybrid connection between rationalism (correspondence and calculation) and empiricism (observation and measurement using instruments) provided a framework for the already mentioned predictability. Paraphrasing Laplace, knowing all the laws of the universe, to "compute" future states depend on computational performance. This statement signals to the other origin of the computer, the need for calculating machines to process mathematical notation symbols requiring some prediction. This configuration, in turn, clearly weaves the relationship between science, language, and the computer, "[i]f science is subject to the workings of language in its conceptions of the real, then is so computer modeling" [Coyne, 1999, 116].

These three connected narratives explain the emergence of the computer. However, although the answer has already been hinted at within its symbolic possibilities, there is still a question that has not yet been addressed: what is so special about the computer? To even attempt a response, the concept of the Turing machine requires further consideration based on the preponderance of progress shaped by the concept of the "universal Turing machine". This supposed universality, a product of a theoretical-mathematical notation construct, gives, at least from the symbolic perspective, an essential set of features that differentiate the computer as a machine from other types of machines. Take, for instance, this statement: "[a] Computer is a strange type of machine. While most machines are developed for particular purposes –washing machines, forklifts, movie projectors, typewriters– modern computers are designed specifically to be able to *simulate* the operation of many different types of machines, depending on the computer's current instructions (and its available peripherals)" [Wardrip-Fruin, 2009, 1] (his emphasis).

It is clear here that the point of inflection that gives the computer its flexibility as a machine is a set of instructions, something that *almost* amounts to saying "software"[14]. I will also argue that the computer's turning point from a *calculating machine* to this simulating *media machine* is, indeed, the emergence of software. In this case, the universal Turing machine can be considered as a theoretical motivation from the realm of symbolic notation already mentioned. This theoretical artifact is amid the controversial hardware-software dichotomy, which obviously began once it became clear

[14]It would be inadequate to consider software as just instructions, but this discussion will be carried out in the next section. Suffice it to say that the instructions respond to the view of the computer as a symbolic machine.

that software was one step beyond the computer as a calculating machine. First, I would like to turn to the classification of computer artifacts made by Dasgupta. In his account, the artifacts fall into one of these three categories: "material" (subject to physical laws, as in the case of hardware), "abstract" (they process and are symbolic structures, including algorithms). The third category, which is called "liminal" (opportunely denoting a state of ambiguity), refers to the most problematic group of artifacts, those that are abstract and material because of their effects on the world. Software can be considered part of the latter [Dasgupta, 2016, 22]. The purpose of presenting this particular configuration is twofold. It shows us an alternative to the hardware-software dichotomy in which software's unstable nature is exposed and enables abstract artifacts to have a recognizable effect outside of the symbolic realm. The Turing Machine is located in this gray zone. It is beyond the scope of this work to provide a detailed description of this symbolic machine, but I will briefly address its origin and the advantages it conveys. The definition of the computer as a symbolic and flexible machine is strongly related to the concept of computability. This concept can be seen as a concern at the intersection of mathematical notation and the rationalistic need for calculation. The famous decision problem (*Entscheidungsproblem*), addressed by Alan Turing in his pioneering paper "On Computable Numbers", can be described –not without oversimplification– as a way of defining whether or not a problem can be solved (computed) with a specific procedure. The hypothesis that establishes this possibility is known as the Turing-Church thesis (the mathematician Alonzo Church arrived at the same result independently). According to this position, "we can divide all possible problems into two classes: those that can be solved by a computer given enough time and memory, and those that cannot" [Eck, 1995, 113].

The importance of calculation should be evident within this framework; solving a problem in the realm of symbols is the ability to calculate its solution. Accepting this hypothesis implies that if something is calculable, there is a "computer" (understood as a symbolic machine) that can do the calculation. Here, the Turing machine comes in as the symbolic and abstract artifact to perform the task. Usually, this machine is described as a theoretical machine consisting of a limitless tape (which moves in both directions) with cells carrying symbols and a read/write head. As mentioned, a thorough description is not plausible here. However, its combination of symbols, writing, and reading operations, as well as its variables and "programmable" states, in short, this *abstract artifact*, provided an approach to the issue of computability based on a simple isomorphic operation. If there was a Turing machine equivalent to the problem, then it was computable. However,

the major step represented of by this notational improvement required by a further abstraction, the so-called Universal Turing Machine. The theoretical value of this lies in that "one can build a single computing machine **U** that can *simulate* every other Turing machine. If **U** is provided with a tape containing the description of the state table for a specific Turing machine, **U** will interpret and perform the same task that a particular machine would do" [Dasgupta, 2016, 28]. In short, the Universal Turing Universal machine granted to one artifact, as an abstract representation consisting strictly of symbols, the faculty to simulate the others. Despite this artifact's relevance for computing, as well as , the computer and software in general, there are some caveats. The first is that, given its strictly theoretical nature, the Turing machine was not intended for implementation. The second is that the modern computer can be more directly attributed to the concept of the stored program, supposedly implemented by the von Neumann architecture for the first time, which was unaware of Turing. Lastly, this abstract machine was better suited to represent non-interactive processes, which are not the norm in modern computing. However, the contribution of this exercise in abstraction can be fully appreciated in modern computer languages that provide a continuum of symbolic equivalences between machine descriptions, from computer machine languages to high-level languages. The mentioned complexity of a system that tends to be ordered through computation, then finds then final abstract realization in computer language-based software, which is what defines the flexibility of the computer as a machine, "[t]his is what modern computers (more lengthily called "stored-program electronic digital computers") are designed to make possible: the continual creation of new machines, opening new possibilities, through the definition of new set of computational processes" [Wardrip-Fruin, 2009, 1].

2.5 Software as Technology

At this point, I have stated two things that, while probably intuitive, shed light on software as an object of study; software is a technology and software is a machine. The top-down approach used assumes that, in this operation, software inherits the properties of technology and the machine as their material manifestation. Consequently, software is not a means to an end; it is not just an instrument. Similarly, and though the idea of a symbolic machine is entangled with the computer, instructions cannot be considered as a manifestation of software's instrumentality; the flexibility of software stems from these instructions. A definition of software is needed to clarify these misinterpretations. This would be a step towards presenting software as a

culture that nurtures its collaborative possibilities. Therefore, we embark on a discussion concerning some of software's particularities, providing a pragmatic working definition. In the process, other highly disputed conceptions about software are addressed, such as its immateriality and opaqueness, to provide further counter-arguments. By defining software, other important relationships like writing and media also reemerge. The complete articulation of these elements will establish a discursive field in which the concept of the community reappears in a less abstract form.

2.5.1 What is Software?

The difficulty of defining software has been widely noted. Therefore, there are multiple and often incompatible attempts. That said, I consider it essential to establish a definition of software. First, I am not interested in a global idea of software. That is not to say that a general construct cannot be offered, but that to acknowledge this possibility –given software's unstable nature– would cause some instances to fall away from the offered description. Secondly, and related to the latter, I am interested in the kind of software that requires computer hardware to be executed. I am aware that the idea of computer hardware can be extensive. In this case, specifically, it points to modern electronic calculating machines, which, as will be explained, surpassed this condition and became media machines. An explanation must be provided here. It can be argued that this configuration reproduces the software-hardware dichotomy; on the contrary, this makes software and hardware not opposites but a tandem where software and hardware are involved in a symbiotic relationship. Although this may seem odd, there is an explanation. I want to avoid making claims concerning software as a merely abstract exercise or even as a metaphysical construct where every process can be seen as software. In doing so, I not only provide a practical limit to my questions, but I also curb metaphysical questions from the scope. This concern will be addressed again when focusing on software as a culture, as I will argue that the metaphysical questioning on the idea of software is a symptom of the scope achieved by the software tropes. This short exercise will begin with basic definitions, evolving into a more complex concept that draws from historical and symbolic assumptions[15]. Once I arrive at a definition, I will address two related issues that often pervade the very idea of

[15]I avoid an etymological approach because I find it more problematic. Although several sources point to a 1958 publication by John W. Tukey as the first use of the word "software", Paul Niquette claims to have coined the term as early as 1953 [Niquette, 2006]. However, I use some of the latter's definition in the discussion on the relationship between hardware and software.

software, its materiality, and its relationship to hardware.

Like many accounts, I trace the origin of modern software to the concept of the computer-stored program; hence, let us focus on it for a moment to find a definition. John Von Neumann's famous "First draft of a report on the EDVAC" is commonly cited as the first appearance of a description of the procedure,[16] which is expounded under a section conveniently called "the code". The following assertion is made when discussing the content and organization of the part of the machine called **M** (memory in what would later be known as the Von Neumann architecture), "[t]his classification will put us into the position to formulate the code which effects the logical control of CC, and hence of the entire device" [von Neumann, 1993, 39]. This passage points directly to the code-operated control of the machine, which mainly governs the central control segment (called **CC**). At this point, one may ask what exactly is "the code" and where is it located? To investigate further, we can go back to the definition of CC to discover that if we want the machine to be "*elastic*, that is as nearly as possible *all purpose*" (emphasis in text), then, the central control unit must coordinate the machine parts to see that "instructions" related to specific problems are carried out [von Neumann, 1993, 34]. It can be inferred from this definition not only that this machine is programmable and can be used for many tasks, but that in doing so, instructions must be followed and are still closely related to the parts of the machine. In other words, the concept of the stored program is intertwined with the computer as a complex machine, which, in turn, reflects the reading of the machine as a system (subsequently, network) of ordered parts drawing directly from technology. To summarize, this "code" can be equated to instructions[17]. Code is a common term to define software, but, as will be shown, it is both insufficient and problematic.

As expressed by Mackenzie in his essay, "The problem of computer code: Leviathan or common power?" "[u]sually, we understand code as a set of instructions that control the operations of a computing machine. Popular and academic accounts consistently reiterate this understanding. It reduces code to a program for a mechanism. We should resist that reduction for a number of reasons"[Mackenzie, 2003, 3]. This formulation immediately re-

[16] I point here to a paper referencing the concept, not its implementation. Arguably, this credit belongs to the "Manchester Baby" in 1948. Similarly, and although the idea of the stored program can be traced to Turing [Copeland, 2004, 17], its treatment is imprecise.

[17] In this part of the historical development of software, the instructions mainly refer to problem-specific instructions. Operating systems and other means related to instructions to control the machine were years away.

calls the ongoing concern about instrumentality and calls for action. Once again, the reasons given speak of an uncharted event-filled territory between written instructions and computing machine operations that limit the discussion to instructions or operations. Despite this observation, this essay solves part of the issue of code, recognizing its dual nature, that code is read and executed. This dual nature, shared by many code and software accounts, partially explains the complexity of the computer machine and its symbolic layers. It even considers code a special kind of language; in fact, according to Galloway, it is "the only language that is executable"[Galloway, 2004, 165]. While, to some extent, I agree with the non-reductionist view of code, I find it more productive to establish a difference between code and software; that is, that they are not the same. In doing so, a hierarchical relationship is established between software and code, making it irrelevant whether code is just instructions or not, insofar as this quality is not transferable to software. Simply put, software should not be reduced. This point obeys to three related considerations. The first one is the *fetishization* of code as a result of business strategies in the 1970s, when software started to be seen as a commodity[18] [Chun, 2011, 19], which explains why code and software sometimes seem to be interchangeable. The second, and from a more intuitive perspective, is that the verb "coding" refers to the very specific activity of *writing instructions* in a high-level language. It may be debatable, but this activity seems to be the more notable within the field of software development (as opposed to design, for example), and there are even development methodologies (such as extreme programming) where that is emphasized[19]. These former two points illustrate why code is confused with software, but not what is missing in a broader conception of software; hence, the third consideration addresses this. If we consider coding as only writing instructions, there are another software-related writing practices which this descriptor does not indexes. Think of documentation, diagrams, and reports etc. Therefore, in order to propose a more inclusive and complex definition of software, we have to think of it as a form of writing beyond computer instructions. A canonical example of such undertaking emerges from the field of software engineering. In his introductory text to the field, Ian Sommerville explains

[18]Here, code is understood as a type of software, not as a part of it.

[19]I would like to point out that this statement (importance of coding) is not arbitrary; it is a reissue of former discussions. If there is a tension in technology between science, design, and craftsmanship in general and the computer emerged thanks to notational advancements, rationalism, and basic tinkering, software has undergone a similar process. As an example, take the book "Hackers & Painters. Big Ideas from the Computer Age", which describes contradictory views of software writing as a science (predictable and mathematically inflected) and as art (prone to trial and error) [Graham, 2010].

that "software is not just the programs themselves but also all associated documentation and configuration data that is required to make these programs operate correctly" [Sommerville, 2011, 5]. Although this statement refers to software engineering than to software directly, as Frabetti observed, the emergence of software engineering as a discipline is based on a definition of software that claims that it is "the totality of all computer programs as well as all the written texts related to computer programs" [Frabetti, 2015, XX]. This statement reinforces the already mentioned point that software is more than code; and in fact, code can be considered a part of software. To approximate a definition, I will contend that software is composed of several artifacts, some them more abstract than others, but all of them sharing the fundamental characteristic of being based on some type of writing (considering writing in the broadest sense of inscription).

I have used the word "system" several times because it reflects software's complex quality, associating it directly with some discussions on technology where an arrangement of elements is necessary to engage in a task. My assessment of complexity is two-fold. From an intuitive point of view, something complex is something difficult to understand. From a more structural perspective, something complex is something made up of several simple parts that interact to achieve complex tasks. It should be noted that software's alleged obscurity and its modern modular nature respond to this wording. This complexity is also observed in the dual nature of code as readable and executable instructions. Now, if we have precisely established that software includes other artifacts besides code, are these also readable and executable? The answer is, yes. In asking himself what software is, Paul Suber proposed several principles of software; I highlight two of them, readability and executability, which, in turn, are a consequence of the pattern principle (software is a pattern) [Suber, 1988].

The readiness of software to be read and executed obeys to grammatical reasons that can be observed when we accept that there is a structure in every pattern. Although this premise could be misleading (because it induces that *everything is software*), I draw two considerations from his thread of reasoning that are suitable to my definition, that there is no data-software dichotomy and that the capacity of execution is not exclusive to machines. As noted, software is in execution when solving tasks, but when it is stored or transferred, it is treated as data; that is, it is read. This makes the dual opposition irrelevant. It can also be said that a software is being executed when the programmer reads a specification (reading a flowchart or picturing the entire procedure without machine assistance) or other written

descriptions –artifacts– that belong to software as a matrix; these artifacts are read and executed. Finally, after drawing from all these avenues, I arrive at a convenient definition of software. *Software is a complex system of communicating artifacts that are organized to solve computational tasks. These artifacts can be material, abstract, or liminal and have a symbolic (written) substrate, which can be read and executed.* I am aware that this is a working definition, and, as mentioned previously, it is not intended to cover all kinds of software. Notably, by recognizing a target in software, it addresses the issue of considering all pattern-derived software. One can find a Turing Machine that can interpret different patterns as a valid language. However, because there is no purpose, these devices are of not of interest, insofar as they are more a haphazard manifestation than a product of organized collaborative efforts. Instrumental assessments or technology are also approached by considering the human-machine relationship. If we consider software as a cybernetic machine, then its possibilities are the product of a close interaction between humans and machines, and not the result of an external tool whose use is only a means to an end.

2.5.2 Software: the quest for materiality

There seems to be considerable concern about the perception of software as immaterial. As briefly mentioned, one of the fundamental purposes of the software studies agenda is to confront the widespread idea of software as an intangible and immaterial subject of study. This misconception is usually construed through the traditional opposition of software and hardware in which the latter is perceived as physical, material, and tangible; in short, *hard*. Intuitively, this perspective does not stand up to general analysis when noticing the material effects of software operations on daily life. However, this approach could be questioned if the consequences of the use of software are considered, not the software itself. Therefore, further examination of the reasons for this particular and problematic position is required, and the concept of software as a machine may be a suitable means to do so.

So far, I have maintained that software is an exceptional machine, which processes symbols (with a physical substrate) assembled of other simple parts. This observation on the contested qualities of the software machine is fundamental; it points to the idea that a machine is visible and reveals the problems of this concept, that is, that software is not visible, therefore, immaterial. If we look at technology's relationship with craftsmanship and manual labor, the idea of machines on a human scale becomes feasible and even part of what is considered normal (machines are made by humans, with

their hands). It can be argued that even the general idea of machine implies references to devices composed of mechanical or electrical parts, a construct presumably related to the industrial revolution. Despite the fallacious reasoning, discussed in previous sections, that consider the industrial revolution the origin of modern technology, its leading role in the development of machine production should not be denied. In support of this point, Heim observes, "...informational systems, unlike mechanical systems in general, are largely opaque, their function not being inferable from underlying, chiefly invisible substructure", where the author draws from the notion of "system opacity" from John Seely-Brown[Heim, 1999, 131]. Stated in this way, and assuming that "informational systems" are related to software (not clearly equivalent), it explains the difference between "mechanical systems" (as commonly a machine is portrayed) and software, the latter having an opaque nature and an "invisible substructure". However, software's alleged invisibility cannot be associated, as I suggested previously, perhaps for the sake of argument, to a simple issue of scale. Think of the growing field of nanotechnology and its machines made of molecules. Although they are invisible to the naked eye without mediation, in my view, they do not lack this material characteristic. It should then be evident that there is something else besides the visibility of the machine parts to which this false immaterial condition can be attributed; this is the symbolical nature of software as a machine. This issue has been addressed in many ways, more than can be mentioned here; therefore, I will mention some of the already almost *classic* discussions on software, language, and materiality. First,a somewhat anecdotal observation can set the tone of this discussion.

In the introduction of a posthumous book by Norbert Wiener –alleged father of cybernetics– he is portrayed as a mathematician obsessed with machines and disappointed with math because of its "lack of the palpable, tactile, visible, concreteness of the machine" [Norbert Wiener, 1993, 1]. This statement goes beyond a personal description; it exhibits the usual opposition in which mathematical notation is assumed contrary to the machine and its physicality. Generally speaking, when physicality is combined with concreteness, abstraction undergoes a similar procedure, with intangibility leading to immateriality. Whether it is mathematical notation, natural language, or computer instruction, the realm of language under this optic creates confusion and the perception of *immateriality*. The reference to Wiener, however, is not gratuitous. According to N. Katherine Hayles, in her well-known book on the posthuman condition, the so-called Macy conferences, held in the United States between the late forties and mid-fifties of the last century –a milestone in the development of the first wave of cyberneticists– are at the

center of an ongoing process in which "information lost its body" (Hayles' well-known expression). This observation acknowledges these conferences as a crucial event in answering "when and where did information get constructed as a disembodied medium?" [Hayles, 1999, 50].

Once again, the use of the medium is key in representing software as a whole, an information machine at the center of a cultural moment defined by "the belief that information can circulate unchanged among different material substrates" [Hayles, 1999, 1]; that is, as an independent non-material layer. Going back to the very nature of language and symbols and following cybernetics precepts where man-machine communication (in the case of software) is achieved through language, another problem arises when considering the relationship between the possibilities of languages in each part of this complex system. "Natural language and thinking in natural language are simply incompatible with the binary digits used on the level of machine language"[Heim, 1999, 132]. As a result, the evolution of software has been the progressive addition of language layers to negotiate the separation between the binary code of the system's hardware substrate and the symbolical nature of the software machine, "[t]he higher and more effortless the programming languages, the more insurmountable the gap between those languages and a hardware that still continues to do all of the work" [Kittler, 1997, 158]. What follows is simple; hardware is obfuscated by a growing load of translations that mediate in software, conferring, on the latter, a condition detached from the very circuits where its commands are materialized. In fact, the semiotic trade of software relies on the conditions of the electronic material where, according to Hayles, "every voltage must have a precise meaning in order to affect the behavior of the machine" [Hayles, 2005, 45]. This observation draws directly from one of the main points of the well-known essay, "There is no software", by Friedrich Kittler, in which he states, "[a]ll code operations, despite such metaphoric faculties as call or return, come down to absolutely local string manipulations, that is, I am afraid, to *signifiers of voltage differences*" (his emphasis) [Kittler, 1997, 150]. In short, software is indeed material, but historical processes and the unstable condition of an increasing symbolic overload give the impression otherwise. Mackenzie has called this issue the "software repressive hypothesis", where "software remains invisible, infrastructural and unspoken" [Mackenzie, 2006, 91].

2.6 Software as a Culture

This chapter establishes the philosophical foundations necessary to understand the twofold methodological approach of a historical description of collaboration and a structural description of the open modes of software production. The discussion takes a top-down approach starting from the construction of the modern idea of technology and taking some relevant elements than can be applied to the software field. This operation is possible because software is presented as a technology; therefore, it inherits its qualities. The following statements are central to the purpose of this work: 1. Technology embodies a type of knowledge that defies old dichotomies that evolved in the science-technology duple with the latter being subordinated to the former; 2. Although technology shares a common core with modern science, technology shows a praxis that cannot be exclusively linked to rationalism and its ways of doing fall on the side of the craft; and 3. Based on the posit of originary technicity, technology has an ontological dimension and is embedded in human nature. As mentioned, these affirmations can also be applied to software. However, a concern arises in doing so; namely, that software is a type of technology, but a special type. But, what does this affirmation entail? First, in order to assess the complexity of software, it is compared to a machine; this means that it is an arrangement of different individual parts (an assembly) that acts as a single system and, in Heidegger's terms, have the potential to perform a task. However, this explanation is insufficient to describe said complexity, considering that software has an additional dimension, a symbolic substrate.

Although the symbolic realm can also become a machine simulator, in short, a symbolic machine [Krämer, 1988], doing so could lead to other problems. This reference above describes the symbolic machine as follows: *"[w]as ist unter einer 'symbolischen Maschine' zu verstehen? 'Symbolisch' meint hier zweierlei. Einmal: diese Maschine gibt es nicht wirklich, sondern nur symbolisch. Sie ist kein Apparat bestimmter physikalischer, z. B. mechanischer oder elektronischer Wirkungsweise, der eine bestimmte Stelle in Raum und Zeit einnimmt, sondern diese Maschine existiert nur auf dem Papier"* [Krämer, 1988, 2]. This phrasing maintains the immateriality of symbols and disregards the electronic substrate indispensable to software. Nonetheless, the second part of the definition states that *"[z]um anderen: diese Maschine macht nichts anderes, als Symbolreihen zu transformieren. Ihre Zustände sind vollständig beschreibbar durch eine Folge von Symbolkonfigurationen, vermittels deren eine gewisse Anfangskonfiguration in eine gesuchte Endkon-*

figuration von Symbolen überführt wird" [Krämer, 1988, 3]. This definition resembles the standard description of a machine that changes states following the instructions on the operations applied to the symbols. This example is given not to point out a possible inaccuracy in what a symbolic machine means, but to highlight the role of symbols in a writing system in which rationalist calculation and the textual part of software could be combined. The construct of the symbolic machine and its contesting abstract-physical dichotomy also highlights the unstable and complex nature of software as a machine. These statements led to the evaluation of software as a flexible technology.

What does it mean that software is a flexible technology? It can be said that this has two main meanings. The first is that the instability of software exhibits a hybrid nature where symbolic and techno-crafting cultures converge. As explained, the manipulation of symbols is the product of the rationalist tradition woven in the concept of computing. Similarly, the emergence of the computer as the machine for making such computations is mainly indebted to the experimental procedures and cultures of engineering rather than to the manipulation of abstract symbols. Secondly, and as a consequence, that software is a complex machine where abstract and physical machines are articulated to make the calculations mentioned. In brief, the software machine can simulate other machines. This complex nature causes software to be misread as a technological artifact, leading to dual simplifications such as those that have been mentioned between the idea of computer science and software engineering as opposite approaches to designing code. The description of software coming from a manufacturing lineage could offer a plausible solution to this complexity, but as discussed previously, this would involve a romantic assessment of technology. Ultimately, craftsmanship, engineering, and symbol abstractions are intertwined into the construct of software. This diversity of origins explains the difficulties in assessing software, among them, its obscurity and alleged immateriality. Of the first one, it can be said that, following the argument presented, software reflects the distinction between technology and technique through the relationship between code and software. In other words, software is more complex than code, although the latter is paramount to its understanding. Of the second, the quest to recover the obfuscated materiality of software is a distinct concern of software studies; to address this concern, materiality should be conceived as complex in itself.

This drives the idea that software presents different specific materialities that are connected to the threads that constitute software as a technology.

It is appropriate here to discuss Boomen's classification of the levels of materiality of software: "1. the materiality of arbitrary media-specific signs (programming languages, interface semiotics); 2. the materiality of non-arbitrary physical artifacts (machines, processors, memory, storage devices, hardware interfaces, cables, routers, switches); 3. the materiality of non-arbitrary inscriptions and patterns (executable code, binary code, magnetic poles, voltage states), and 4. the materiality of non-arbitrary digital-material metaphors (objects, tools, places, commands)" [Boomen, 2014, 151]. These levels again manifest the complexity of software, describing the different types of materiality that can be found in developing, using, or studying software. As Chun said, "[s]oftware is, or should be, a notoriously difficult concept. The current commonsense computer science definition of software is a 'set of instructions that direct a computer to do a specific task'. As a set of instructions, its material status is unstable; indeed, the more you dissect software, the more it falls away" [Chun, 2005, 28]. On the one hand, this statement can be interpreted as an affirmation of the mentioned complexity and instability of software, but more so, as an acknowledgment of how software's different materialities are unstable.

A conclusion must be drawn to end this summary and the reflections contained in this chapter. Software is indeed a culture, based on software's complexity and the different historical moments and developments that stem the core of its creation. In other words, software culture encompasses the practices, traditions, common tools, abstractions, customs, conventions, and metaphors that are shared, discussed, and problematized by the community that represents the living part of that culture. A corollary of the last paragraph is that this culture exhibits several layers of materiality. To expand on the discussion of software as a culture, I will focus on two main points, the precedent of the algorithmic culture and the materiality of metaphors. Algorithmic culture is of interest because algorithms are at the base of computer science, programming, and software at large, although it precedes them. Like software, algorithmic thought exhibits a composite nature because, on the one hand, it offers a practical method to solve problems by following steps and, on the other, through history, algorithms have progressively employed symbols to achieve their goals [Ziegenbalg et al., 2016]. This places the idea of rationalism, symbols abstractions, and their consequence in computing (calculation), ergo software, into another perspective. However, the algorithmic culture partially implies the creation of a symbolic machine, understood not only as the mentioned flexible machine but as a symbolic machine that presents the mentioned levels of materiality, in what can be interpreted as a hybrid object [Stach, 2001, 3]. This concept of a hybrid object refers to

abstractions that use formalism and notations but behave and operate as a machine. This hybrid nature impacts software culture by prompting changes within its evolution and showing the tensions that have been misinterpreted under the tenet of the immateriality of software. To exemplify the last assertion, consider the story about the computer scientist Dijkstra and his quest that pursued turning programming from a craft to an engineering discipline. Such a tasks engendered structured programming, which Dijkstra pioneered "precisely because he began his programming career by coding for machines that did not yet exist" [Chun, 2005, 37]. This quotation highlights this hybrid nature, where new developments can be obtained from symbol abstractions. Lastly, metaphors must be assessed to broaden the construct of software culture. Metaphors are present in software culture in two forms. First, they are present as borrowed images to describe software features and possibilities when there was no previous reference to explain them (think of Desktop, client/server, web page, etc.) or second, as translations of software culture elements into other fields. The latter outlines this work's research inquiry by showing the impact of the software culture, which has been used to describe our brains, the universe, and the functioning of cells, serving as the metareference of our times. In the next chapters and following the proposed methodology, historical and structural accounts will explain how one of the primary digital-material metaphors of the software culture evolved –collaboration– and how it is represented in the modern modes of open software production through the human-machine networks.

3 — A modular history of collaboration in software

3.1 Introduction

Up to this point, software is considered a culture whose tropes point directly to several aspects of this technology as a symbolic machine, a type of writing, and an unstable and complex artifact. The electronic digital computer has come a long way from its creation as a calculating machine, and its emergence as a simulation and universal media machine, and software has been a fundamental agent of this change. Software is responsible for increasing the former calculating machine's flexibility. Its prominence is so notorious and its historical development so meteoric that in little more than 50 years, it has gone from being almost non-existent –even nameless– to become one of the greatest industries of our times. Having said that, a historical approach is taken to explain the evolution of software, particularly through the lenses of open collaboration in software development processes. However, some assumptions have to be made in order to do so. The first one is that the history of software is not linear. Although this statement could be considered obvious when looking at any technology, software has some unique attributes that reinforce this assertion. Among them, the already mentioned non-linearity of software's written substrate, the implications of the Derridean view on the line as a model of historical claims, and the departure from the corporate-driven hardware and software industry reflected by the creation of the personal computer (and its hacker and counterculture roots). Another factor could be the ongoing and unresolved tension between programming as a science or an art, which has been completely subdued by the efforts of software engineering as a discipline. The second assumption reestablishes the concern regarding the immaterial misunderstanding and corroborates the entangled nature of the software and hardware tandem. Though usually presented as a layered and well-defined composite (with software at the top), the depen-

dence of software advances on very physical and almost prosaic hardware issues constantly reminds us (but apparently not effectively enough) of the fallacious nature and drawbacks of the material/immaterial opposition. The third assumption concerns the complexity of historical narratives in which various forces coalesce. From the perspective of software development, although sometimes devised in an area of the discourse, each open collaboration follows a series of tensions where politics, entrepreneurial pragmatism, and even idealism highlight the cultural relevance of these characteristics.

I am now able to express the central hypothesis behind this chapter. Software is an original collaborative technology and, despite its elusive history, several events confirm this feature. Although the emergence and popularity of today's open collaboration software metaphor owes a great deal to the FLOSS movement, which made it a popular topic of discussion in new media theory, there are several examples of open collaboration in software development that predate it. The importance of FLOSS was that it discursively organized and compiled a set of already flourishing practices in the industry into a few understandable principles. It must be noted that despite the efforts of several historical accounts, the unstable nature of software and its non-linear history has been challenging to such endeavors, casting some doubt on how specific processes and assumptions are perceived. To illustrate this last statement, consider these two positions. Campbell- Kelly, a specialist in the history of the software industry, claims that collaborative practices already existed in the IBM corporate culture, based mainly on a pragmatic concern, and that these practices resemble an open-source model [Campbell-Kelly and Garcia-Swartz, 2009]. On the other hand, Eric Raymond, a notable figure from the open-source movement, pointed out the inaccuracy of portraying the hacker culture and FLOSS as an idyllic "pre-proprietary golden age", which certainly was not the case [Moody, 2002, 145]. These narratives are not contradictory, and by challenging a corporate versus activist dichotomy, they show how problematic it is to offer even a partial picture of a fragmented non-linear history. The historical events that will be described were selected to provide a perspective on the evolution of open software collaboration, and in no way constitute an effort to provide an all-encompassing narrative on software, which would be nearly impossible.

The selection of the historical facts and events that make up this chapter are closely associated with four interwoven threads that underlie the narrative. The first one is the presence of a cybernetic reading of some of software's advances and its collaborative patterns. In the process, some phenomena, such as the advent of interactivity or modular programming, find a parallel

theoretical framework that, in turn, provides a picture of the environment in which they took place. The second is a reference to the free/libre open source software movement. Though ubiquitous and frequently mentioned, here, the idea is to offer a context of the FLOSS effort, not only as a cornerstone of open collaboration but also as a re-edition of some past practices theorized under the normal interaction of several earlier forces very present at the moment of its emergence. The third is the persistent reference to system tools (operating systems and programming languages) as a prominent type of software where collaboration takes place but also as the direct instance of a type of software located precisely in the problematic software/hardware juncture. Last but not least, software engineering is discussed. The importance of software engineering is paramount in understanding the growth of the software industry and the tension between various interpretations of how software should be produced. Two final observations accompany this four-sided position. So far, I have not provided a proper definition of *open*, which I have certainly left open, awaiting the segment on FLOSS to attempt one. This is out of a desire to allow different historical conceptions of this *openness* to be discussed without preconceptions. The sections, although organized by thematic descriptors and not time, follow a rough sequence. It begins with "collaboration prehistory", which covers the 1950s, "interactive dreams/compatibility saga" covers the '60s and '70s; and FLOSS, the '80s and '90s. However, the section on software engineering, which begins in the '60s, runs parallel to the other sections. There have been many innovations in recent years (particularly in the mobile environment), which are considered shaped by previous events; therefore they remain the focus of this chapter.

3.2 A collaboration prehistory?

3.2.1 Before there was software

Before moving forward with a full discussion on open collaboration in software, I think it is worth looking at some notions surrounding the development of the electronic computer. A question concerning the explained instrumental issue may ensue when considering openness and collaboration[1]. While it is clear that some modern computers encourage open collaboration (as FLOSS

[1]For now, I will use the two rather intuitive notions of *openness* and *collaboration*. The first is about access to information, that is, the extent to which designs, advancements, blueprints, etc. are available to other parties for reading and even building upon. The second is more about a process and how a workforce is organized and whether the efforts go beyond organizational boundaries.

has shown), could this statement also apply to the first electronic computers? Furthermore, were these early computers designed and implemented following a pattern that, even if a little, could be described as *open* or *collaborative*? The first question has already been answered in the popular disquisitions of celebrated pioneers that prefigured some computer-related tropes. Just to mention a couple, think of Norbert Wiener and his cybernetic-based belief in the entropy of information and its tendency to flow without titleholders, which reinforces the idea of cumulative knowledge, based on circulation and feedback [Brate, 2002, 28]. Similarly, Vannevar Bush's widely mentioned "Memex" describes a never built electromechanical device conceived in part to share information [Brate, 2002, 42]. These thoughts fall directly into the idea of the free flow of information and machines that support this need. The second question brings forth a more complex idea of the computer machine, not just as an instrument to open collaboration, but as a field of experimentation for various practices that would later constitute the metaphor of open collaboration in the age of software. This entangled relationship once again reveals the closeness of hardware and software and the mutual interdependence of their evolution.

Some comments on open collaboration processes can be briefly addressed concerning the appearance of the electronic computer itself. On the one hand, the creation of the hardware required a coordinated effort in which the mentioned idea of information flow must have been considered in the context. On the other, one of the lines behind the computer, scientific inquiry, also exhibits patterns of open collaboration, according to an alleged academic tradition[2]. First, we need to discuss the emergence of the electronic computer properly. It is beyond this section's scope to address whether some of the precursors of the digital electronic computer exhibited some of these desired characteristics. For instance, Pascal and Leibniz's mechanical calculating machines or Babbage's more recent analytical machine are examples that are vastly used in historical accounts on the computer. Even reducing the scope to the *modern electronic computer* makes this a baffling undertaking. Although some advances were made in the 30s, the early electronic digital computer gained its highest momentum amid the war efforts of World War II, where it had, obviously, secret implications [Ceruzzi, 2012, 21]. Its specific use in ballistic calculations and cryptography made the development of the computer seemingly incompatible with open or collaborative objectives. However, some observations could be made to attenuate this inference. Take,

[2]However, this relationship is not seamless and there are too many elements to consider within the interaction under the open-access movement [Hall, 2008, 155].

for example, the shift from calculations made by humans (called computers) to this machine (electronic computer), one of the multiple pragmatic motives of its conception. It follows that a complex calculation had to be divided into sub-operations to be performed. These sub-operations were distributed among *the computers*, requiring the coordination and cooperation of the human computers to arrive at a cohesive and correct result[3]. Summarized in the words of Comrie, one of the pioneers of the humans-based Scientific Computing Service (a long-standing service, overthrown by the digital computer), this type of human-based computing was simply "a question of organization" [Campbell-Kelly and Aspray, 1996, 67], between the women who performed this work. Although simple, this observation indicates some requirements in calculation, such as the ability of allocating problems, and branching, among others, that were carried over into the very design of the early electronic computers. In some ways, this reminds us, because mechanical calculation machinery assisted human computers, of the systemic man-machine network at the heart of these developments. Another clearer and less organizational instance of collaboration, again from the war, is the computer called "the Bombe". This joint venture between British mathematicians and American engineers was used to decode German war messages [Ceruzzi, 2012, 36].

Second, the idea of scholarly and open collaboration patterns can be seen under the light of the development of the early electronic computer. After the war, by the late 1940s, the *zeitgeist* was different; it was flourishing with discussions of computer designs among the academic community (mathematicians and engineers)[Ceruzzi, 2012, 29]. Specifically, the idea of *community* and the crossroads between the early computer and the academic tradition had one important antecedent in the figure of John Von Neumann, whose famous and unpublished "First Draft on the EDVAC" (an influential computer design with an early description of the stored program procedure, a forerunner of today's software) was mimeographed and distributed for comments reflecting "the openness of academic-oriented scientists"[Isaacson, 2014, 111][4]. The fact that Von Neumann (or, for that matter, the group behind the ideas he put on paper) draws from Turing's theory of the Universal Machine, who, in turn, was busy with his own computer design (The ACE) depicts an academic environment of feverish activity in pursuit of the electronic and

[3]It can be argued that even when a system of calculations is too complex to be solved by humans –no matter how many– the idea of breaking a problem down into smaller steps lies at the heart of the algorithmic thinking and the computer.

[4]Ironically, while this gesture can be considered a display of "openness", the ensuing disputes on patents and the prominence of the Von Neumman name on various collective ideas, reveal a not so collaborative effort [Godfrey and Hendry, 1993, 1].

programmable computer. To conclude, and although such an idyllic portrayal of the academic community can be considered *romantic*, the modern movement called "Open Access" can be seen as the modern realization of such scholarly altruism where the authors are seen as "a tribe" sharing information [Suber, 2012, 2], whose statements clearly indicate a relationship in which "[a]n old tradition and a new technology have converged" [Budapest Open Access Initiative, 2002].

As seen, the creation of the computer was a fruitful field of innovation and ideas closely connected with software tropes as a symbolic machine that allowed and embodied open collaboration. However, to transition from the computer to software, it is convenient to look, even superficially, at some of the innovations that made the symbolical versatility of software possible, even from its obscure beginnings when it was nameless, let alone had a *community* around it. The first innovation was the decision to put instructions and data in the same space. Although today this can be considered a logical design option, as usual, the origin of choices obeys multiples causes. For example, Howard Aiken's famous electro-mechanical computer, the Harvard Mark I, in service during the war, used different memory spaces for instructions and data for the sake of integrity [Cohen, 1999, 159]. However, as observed by Ceruzzi in his account of the computer's early developments, the difference between data and instruction intensive applications became evident (even at the early stages), favoring the more versatile approach of storing instructions and data in the same memory space [Ceruzzi, 1997, 6][5].

The second major development was the implementation of instruction branching. Rooted in the requirement of breaking down large calculations into more manageable procedures, instruction branching was implemented to provide some designs the ability to jump from one instruction to another nonconsecutively. Addressable memory space was a major leap in computing that broke the linearity of instructions processing inherited from abstract artifacts, (such as the Turing machine, and instructions using a punched tape)[6]. Lastly, and perhaps the most significant advance was the creation

[5]This decision is directly related to software's ability to be considered instructions or data in a same space or channel, a feature that associated with the role of software in networks.

[6]An interesting fact must be clarified here. As branching was introduced, a change from parallel to serial processing was also made to facilitate programming, that is, "computing lines rather than columns" [Aiken, 1964, 66]. However, Von Neumann's architecture built a bridge separating instruction storage from processing, which made possible a non-linear feeding of instructions into the serial processing unit in a *one by one* fashion, that is, serial

of the stored procedure program. Like the other concepts and the computer in general, no single event has marked the introduction of this often mentioned precursor of software. However, it can be argued that regardless of some theoretical substrate, it was pragmatics that played a key role in the emergence and implementation of this feature. More precisely, the so-called "Newton-Maxwell gap" [Ceruzzi, 1997, 5], which determines the speed between mechanical instruction feed and electronic instruction processing, demanded a solution to keep the vacuum tubes of early computers as busy as possible, taking advantage of their processing speed compared to other mechanical devices [Cohen, 1999]. The approach given was then, the loading of instructions into electronic storage before being executed electronically. As explained, the division of storage and processing present in the Von Neumann Architecture, design of undeniable influence at this early stage of the electronic computer, allowed the change to be made with an unpredictable yet enormous impact.

To summarize, despite the obscurity of this initial period, several examples have revealed a more pragmatic-driven development of the computer where a founding academic community revolved around its theoretical potential, achieving important advances that would define not only this machine but also the coming of the Software age. Moreover, key hardware developments in this pre-software era paved the way for some of the most notorious qualities of computer applications (and the introduction of further advancements in the symbolical realm) that yielded, after some years, the flexibility of its possibilities and prompted the open, collaborative practices in its deployment.

3.2.2 A collaborative PACT for a language

Although determining precisely when software emerged is challenging, the conceptual importance of the stored program as a kind of *proto-software* seems general understanding. Its properties displayed some of the qualities expected from modern computer programs; in fact, it can be asserted that without its development, modern computing would have been impossible [Cohen, 1999, 270]. As discussed in the last chapter, the word "software" was barely mentioned in the '50s; the first printed evidence of the word dates back to the end of the decade. The term came into fashion well into the mid-sixties [Campbell-Kelly and Garcia-Swartz, 2015, 34]. Thus, we can say that indeed software existed in the '50s the fifties, but scarcely. This early stage

[Ceruzzi, 1997, 7].

in the history of computing emphasizes the previous statements on how software and hardware cannot be thought of separately and how their evolution is intertwined. Moreover, the interaction between materiality and symbols already existed in the stretch of developments of the burgeoning mainframe industry of the '50s, embodied in the emergence of computer languages in the hardware industry.

Building on this perspective, I will succinctly follow a narrative to argue that collaboration was already present in this shifting movement when former technologies gave way to the electronic computer, and specifically, the software age was born. Namely, I will discuss an early collaborative project for an experimental compiler called PACT. However, two considerations must be made to frame the latter statement within the overall plan of this work. First, the collaboration at this stage was related to the academic tradition discussed in the previous section. PACT was an example of a pioneering enterprise in which a massive calculating field, such as the aviation industry, played an important role. It could also be considered as a precedent for the SHARE community, an IBM-related software exchange community that operated on principles that can be traced back to collaboration in engineering standards and academic exchange [Akera, 2001, 734][7]. The second consideration is the relevance of hardware, not only because of its mutual dependence on software at this early stage but to highlight IBM –a hardware manufacturer at the time– as the most popular computing platform among participants of the PACT effort, and the default choice of the collaborative tasks to be described. This appreciation is reinforced by tracing a collaborative tradition in other earlier calculating machines, such as the CPC (Card Processing Calculator, invented by IBM) and punched card equipment in general. Although some configuration designs (established hardware wiring designs) were used and shared by late users of CPC technology, according to the innovator Paul Armer, the cooperative opportunity was missed in this field [Armer, 1980, 124]. Similarly, in more general terms, a direct precedent can also be found in the field of machine accounting, where users shared and exchanged wiring diagrams using distribution channels such as a newsletter ("Computer News") or the manufacturer's newsletter (IBM's Pointers) [Armer, 1980, 123,124]. Some of these examples, even stress the price factor involved in collaboration (and the very physicality of programming) and a broader picture in which reciprocity in knowledge sharing was expected,

[7]The importance of the aviation industry is evident when one considers that the PACT project working committee was composed almost entirely of companies from the aviation sector [Melahn, 1956, 270].

"[a]chieving co-operation in this case consists of convincing a fellow worker that he should supply you, gratis, with a set of board wiring diagrams or some similar piece of know-how in the hope that someday you may have something the lender can use" [Melahn, 1956, 266]. Looking at these threads, one could say that a project like PACT appeared as part of a subtle collaborative tradition in calculating machines, new evidence that "cooperation is the greatest invention since the wheel" [Armer, 1980, 125]. However, if the above statement is true, the collaboration effort represented by PACT is on another level, where the symbolical layers between man and machine were still new and promising.

PACT stands for Project for the Advancement of Coding Techniques. It could be an example of a joint venture in the mid-1950s in which several companies used computers to facilitate the way their business operations were conducted, developing software, explicitly, to comply with their needs. The concept of PACT as one of the first collaborative software efforts reveals the lack of software at the time, an era when the very concept of the computer application was still taking shape as a major force. It also points to the diversity of available solutions being implemented to fill this gap. When a company decided to purchase or lease an electronic computer to aid its processes, the machine delivered was pretty much just the raw iron. It barely included "the user's manual-IBM's were called Principles of Operations - a crude assembler, a loader (non-linking), and a few utility routines" [Armer, 1980, 122]. Therefore, the companies that had this kind of technology were responsible for the development of their own software, a trend that continued even into the 1970s [Campbell-Kelly and Garcia-Swartz, 2015, 35]. This in-house model implied, on the one hand, the modification of a few applications bundled with hardware and, on the other, establishing communities, namely, user groups aimed to share information on a specific platform[8] [Campbell-Kelly and Garcia-Swartz, 2015, 35]. Focusing on the issue of software development, two main drivers have been considered crucial in the genesis of PACT, software development costs (in time and money), and the trade-offs between human developed and computer-aided code. Writing software applications in the mid-fifties was an expensive enterprise; it still is today, with troubleshooting times taking up to 40% of the project time with costs ranging from $2 to $10 per instruction [Melahn, 1956, 266].

[8]The ability (legal and technical) to change existing code and the sense of a sharing community somehow resemble with decades of anticipation the values and procedures incorporated by the FLOSS ethic. I also overstated the idea of community as an indispensable part of a technical system.

Moreover, in the '50s, a shortage of workforce required tools to automate some of the programming activities. However, their underdevelopment affected the quality of the produced code, which was far from optimal, resulting in a machine use overload, let alone the time used to develop a diversity of assistance tools and routines libraries and their incompatibility [Melahn, 1956, 266]. In 1954, and as mentioned, with a strong push from the aeronautics industry, the idea of a collaborative effort emerged among some west coast companies in this fertile scenario. The PACT initiative encompassed two projects, the PACT I and the subsequent PACT IA, a division closely connected with the hardware in which it should operate. Here, it is necessary to stress PACT's dependence on the platform selected by the partners involved, the IBM 701 and its successor, the 704. These early IBM mainframes were originally intended for scientific applications [Campbell-Kelly and Garcia-Swartz, 2015, 18], a clear manifestation of the historical milieu in which commercial applications had not yet taken over the software industry. The incompatibility between the two models only contributed to the existing Babel tower of user-generated software. The companies behind PACT decided to begin (justly, as we will see) programming the 701, although the shift to the 704 had already been planned. In response to the mentioned lack of software in these early mainframes, there was a proliferation of user applications (modified or new) that profited from the fact that IBM delivered its code in both binary and source to manage the insufficiency of their software offer [Campbell-Kelly and Garcia-Swartz, 2015, 34]. To sum up and return to a previous statement, this was a scenario where a cooperative effort was more understandable from a very pragmatic perspective.

Given this outlook, the enthusiasm surrounding the PACT project, and the automatic help tool being developed collectively was no surprise. However, and for different reasons, the tool did not reach the expected scope, in the end, it was outdone by other approaches. Nevertheless, this effort displayed several features that can be considered as precursors of collaboration in a modern software project, and in doing so, several novel and even software engineering techniques were fulfilled. To explain the last statement, we can briefly look at how the project was organized. PACT was organized into two committees, the policy committee in charge of administrative decisions and the working committee for the technical aspects [Melahn, 1956, 267]. These committees represented members of all the companies participating in PACT. In their effort to communicate and make coordinated decisions, they overcame language differences and their own competition-based business culture [Armer, 1980, 125]. It could be said that PACT was not only about companies sharing code. It was a coordinated enterprise in which

each participant was responsible for a part of the project, and tasks such as "subroutine" "linkages", "card reading", or "loop expansion" were assigned among the members of the committees [Melahn, 1956, 270,271]. The following case can be considered as an example of the administrative and technical agreements. PACT's design plan reflected an approach to code reuse that was very rooted in the technological pragmatism most present in the early years of a novel technology.

On the one hand, the project set out to hastily explore other available tools and languages in order to preserve coding energies (notably, UNIVAC A-2 compiler, Whirlwind programs, and even the developing FORTRAN) [Baker, 1956, 272], [Melahn, 1956, 268]. On the other, and remarkably enough, the project decided to start codifying the 701 knowing that it was about to be retired in favor of the newer, faster, but incompatible 704 [Melahn, 1956, 267]. This seemingly controversial decision can be understood by considering that experience obtained from the soon to be discarded 701 could be used to configure the 704 more suitably. Furthermore, when the successor for the PACT I, the PACT IA, intended for the IBM 704 mainframe, was being developed, it was directly influenced by the know-how obtained from the previous platform; this provided an additional opportunity to correct previous errors and refine the language [Steel, 1957, 9]. Here, we find an indication of an early incremental model born out of hardware limitations. The described corporative and technical collaborative decisions show how a seemingly difficult path gains new light when more businesslike criteria and planning are included in the designs. Some technical achievements can be mentioned to support the statement on the advantages of the collaborative effort from the technical perspective, as a more in-depth discussion would be beyond the scope.

Given the early stage of computer languages, computing, and the field of software at large, the advances of the PACT I and the PACT IA compilers bring forth some technical refinements of remarkable importance. For example, the PACT I implemented an early modular design in which blocks of code where organized by "regions" and were called by a control region, which could also be controlled by a higher region, giving a top-down approach to problem-solving [Baker, 1956, 272]. The concept was not invented by PACT; instead, it was a choice made after studying other systems; this points to the already mentioned committees' tasks. PACT, as a symbolical layer, similar to a machine but simpler, obtained a notation closer to algebraic expressions that reduces errors in arithmetic programming and scaling (which represents 15% of errors), as well as bringing the user closer to the

system and avoiding some machine code dependencies [Miller and Oldfield, 1956, 288]. Also related to the language of PACT, variables with mnemonic names and not with fixed addresses, conventions that are taken for granted today, were recommended as desirable for the PACT I [Baker, 1956, 272], with some improvements, they were implemented in the subsequent PACT IA [Steel, 1957, 8,9].

In the end, the PACT project (with its two launches, PACT I and PACT IA for the IBM 701 and the IBM 704, respectively) was not destined to be a lasting enterprise. It was, in fact, overshadowed by other initiatives, notably, FORTRAN. Although PACT compilers appeared at a time when computer languages were still an exploratory field, their impact on further developments cannot be adequately assessed. It can be argued that hardware constraints prevented the PACT compilers from becoming more influential. The committees knew that a more abstract language would ease the burden of programming, but, at the same time, this would require more processing time for translation; therefore, they decided to leave PACT close to machine language. As a result, the FORTRAN approach, which was more on the side of mathematical expressions, succeeded, driving a major change in the software industry embodied by the introduction of more symbolical layers to veil the machine code (a path taken when considering the popularity of high-level programming languages). FORTRAN facilitated the process, obtaining more considerable popularity [Bemer, 1982, 814].

PACT's importance is its collaborative nature; it is an early case of applied cooperation in software development. The pragmatism of the PACT focus, which is a recurrent trope of collaboration in software and technology in general, proved its value in its figures compared to previous approaches. PACT I was fast; it reduced the problem-solving time from days to hours and produced fewer errors, which, if they were encountered, were easy to find and correct [Greenwald and Martin, 1956, 309]. Beyond the pragmatic benefits of the products of this type of collaboration between early adopters of software, the development process itself would be profoundly affected by methods that not only resonate with the systematization of the field of software engineering years in advance but that sound very familiar even to FLOSS accounts and recommendations. To illustrate the last point, consider this anecdote: "[o]ne of the members of the working committee programmed a routine to extract the square root of a matrix using an iterative technique. Coding time was one hour. He compiled the code in three minutes (some 50 PACT instructions generated 500 machine language instructions). On trying the program he discovered one error he had made and an error in one of the sections of

the compiler. He phoned the person concerned with that section and a few hours later received a return call with the correction needed. He recompiled and ran the job which, by the way, had never been successfully programmed at that installation. Total elapsed time: less than one day" [Greenwald and Martin, 1956, 310]. This quotation suitably sums up the spirit of the PACT project and its qualities as a socio-technical matrix where communication and coordination between companies overcome the isolationism that characterized the earlier IT efforts of the first actors in the software field (still to be named).

3.2.3 A SHAREing community

The PACT project was a direct precedent of a more ambitious initiative, which stills exists today in a different form, the SHARE group. This community of IBM machine users, established in the mid-1950s (1956), in the mainframe era, is allegedly the first user group of the history of computing. However, the move from the PACT experience to a user group like SHARE was not immediate. It was a kind of logical move considering that there were not many computer installations at the time (therefore, the companies behind the PACT project also participated in SHARE) and that both occurred almost simultaneously. As Paul Armer explained in his cited 1956 talk in the first year of SHARE, "[t]his much of SHARE genesis was no accident it flowed naturally from the PACT experiences"; but the accident was a meeting of IBM users that helped to spread the enthusiasm [Armer, 1980, 125]. Both experiences have similarities; however, two are essential from the software perspective that is developed throughout this work, hardware dependency, and pragmatism. The first can be considered as another sign of software materiality and how it should always be considered concerning hardware. Like PACT, SHARE is strictly based on IBM installations, specifically on the scientifically oriented 701 and 704 mainframes. However, while PACT is the independent initiative of users whose common hardware platform –not software– was based on these mainframes, SHARE was closer to the computer manufacturer. It worked as an interface between the users and IBM, providing a communication channel to share software and hardware improvements. IBM's prominent role should come as no surprise, considering its dominant market position in the 50s, when the industry was called "IBM and the seven dwarfs" [Campbell-Kelly and Garcia-Swartz, 2015, 12].

The second involves pragmatism and the high costs of software development; the considerable need to make cost-effective use of this new computing technology was at the center of the motivations. The cost factor was "hor-

rendous", it represented "in excess of a year's rental for the equipment" [9] [Armer, 1980, 124]. For its members, SHARE represented a kind of win-win situation; member companies benefited from a growing library of computer applications, and IBM could promote its hardware by adding these applications to its packaged computing offer. Beyond pragmatism, and more abstractly, SHARE was a community in which cooperation was a major asset in the collective pursuit of the improvement of key technological abilities, which were scarce at this early stage of computing; "[t]he collective knowledge and wisdom that had been accrued by them was quite limited. Thus, cooperation in those early days was more necessary than it is today to make effective use of data processing system" [Armer, 1980, 123]. The user group was very aware of the knowledge-based nature of its goals and even traced software collaboration back to the academic tradition. As attested by a statement in a SHARE document dating back twenty years after the foundation of the user community, "[t]he principal purpose of SHARE Inc. is to foster the development, free exchange, and public dissemination of research data pertaining to SHARE computers in the best scientific tradition" [Share Inc., 1977, 2]. To summarize, although similar in their urgency, SHARE stands above PACT as a more challenging project. It was not only a compiler initiative but also addressed software in general, and in doing so, it built an organization that promoted early open collaboration on software, coordinated the corresponding activities, and managed to remain an independent association with strong ties to IBM.

According to Campbell-Kelly, IBM's development model of the 1950s could be considered *open source* because "its software source code was open, free of charge, and written collaboratively with its users" [Campbell-Kelly and Garcia-Swartz, 2009, 229]. This statement is based on the computer manufacturer's sponsorship of SHARE as a way to obtain system tools,[10] and specifically, of the SHARE community organization, which is said to have predated the practices of the open-source community by decades [Campbell-Kelly and Garcia-Swartz, 2009, 230]. From a workflow perspective, SHARE displayed a type of collaborative software engineering approach. A wish-list

[9] It should remembered that at this early stage of computers, their purchase cost was so high that leasing remained as the only possibility for many companies still experimenting with the technology.

[10] As computing advanced, a differentiation between common *system tools* and business-oriented *applications* became evident. Apparently, SHARE devoted its efforts to the first, not only because system tools were needed by all the members but also to avoid antitrust issues, given that the members' core business was far from computing. As mentioned, the aviation industry played an important role in these initiatives.

of required software was created to allocate the programming efforts among the members; the submitted software was checked for quality assurance before finally arriving at the shared library [Campbell-Kelly and Garcia-Swartz, 2009, 230].

Standards and internal organization are two aspects (among others) that are well entangled into these procedures, supporting them but also being shaped by them incrementally. Standards are more involved with equipment nuisances; they emerged as technical requirements to find common ground, given that hardware incompatibility was the norm. The second, internal organization, involves the human component and the coordination between companies. Both can be considered as language negotiations apparently influenced by pragmatism. On standards, several examples establish the positions, one corresponding to a more symbolic level (mnemonic operation code, an assembly program) [Armer, 1980, 126] and the other, a more material substrate (standard punched card format, print wheel configuration) [Akera, 2001, 717]. Internal organization prompts the relevance of the community within software's socio-technical system; therefore, it involves more complex interactions. Let us consider, for example, a prominent topic: competition. Some SHARE companies were business rivals; therefore, cooperation was not a seamless process. There was also an inherent lack of confidence in the technical abilities of employees belonging to another corporate culture. However, as the benefits of cooperation became evident, a more respectful attitude developed. In fact, "[a]t the first meeting of SHARE, disdain for the other fellow's abilities was gone – there was general 'agreement to agree'" [Armer, 1980, 126] "common goals and technical interest overtook corporate loyalties" [Akera, 2001, 718].

Once these *corporate loyalties* were overcome, the ongoing collaboration ethos followed patterns that differed from the normal corporate vertical communication. According to Akera, "concerns about hierarchy often gave away to a sense of open participation among the delegates" [Akera, 2001, 717]. This somehow evokes –again, decades in advance– the alleged and celebrated (yet problematic) horizontality of a FLOSS project. Finally, there was the question of new members. SHARE was an open user group; by the end of its second year, "[a]ll new IBM 704 installations became SHARE members" [Akera, 2001, 720]. According to the "Obligations of a SHARE Member" section of the SHARE user manual, members had an obligation to maintain the "cooperative spirit" and have an "open mind" in discussions. They were also advised to observe the standards and always carefully negotiate when to diverge from them slightly [Edson et al., 1956, 8].

In conclusion, the described characteristics of the standards and the organizational issues of the SHARE approach could be effortlessly be read on a FLOSS key. However, there is still one piece missing, the software project itself, a SHARE user group collaborative effort. The system for the 709, the SOS (Share Operating System) was precisely that, particularly, the part called SCAT (SHARE compiler and translator).

The SOS project illustrates the importance of language in understanding the nature of SHARE's cooperative effort. Concerning standards, language interactions can be viewed as two-fold: as an agreement on the internal language of the machine (a great achievement, considering that during this decade, even the bit in which to start the decimal part of a number was matter of discussion), and as a definition of the logistics of meetings, mail tones, and other idiosyncrasies of group communication. SHARE, however, had already settled on the so-called SHARE assembly program as the communication (code) standard for distributing programs. This assembly program shows software's essential incremental nature. It was a contribution from a member of the user group (United Aircraft Corporation) that had modeled it after the original 704 assembler, which, in turn, goes back to the "Comprehensive System", used in the seminal M.I.T Whirlwind computer [Shell, 1959, 123]. SHARE's organizational communications device was driven (similar to the case of PACT) by the imminent arrival of a new platform, the IBM 709. Its arrangement, which had an obvious connection to the 704 platform and a substantial library of programs –with some 300 programs being "distributed to the membership" [Armer, 1980, 127] made plenty of sense, given the incompatibility issues in preparing the new machine. To this end, the SHARE committee established a 709 committee. The result of this committee's initial work was the SCAT, an integrated solution to programming (as its name may suggest, a compiler, assembler, and translator) that aided programming, debugging, and overall human-machine interaction. The last statement highlights the increasing complexity of computers and programming. Fittingly, to achieve a breakthrough in "higher language coding situations", it was necessary to make "the machine reply to the programmer in the same language that the programmer originally used to code his problem" [Shell, 1959, 124]. SCAT attempted to accomplish this by introducing a new layer, a symbolic machine language intended to serve as a communication standard for the inputs and outputs of various programming processes. It is worth noting, that this human-machine arrangement not only presents programming as a complex system, informed by interactions between the parts –a sociotechnical system– but that, even though the human represented is the same

person (i.e., the programmer), the possibility of other people participating in the process make it open, implying the potential for collaboration and "enhancing" program communications within SHARE through this common language [Shell, 1959, 127]. Although SCAT was evidently the most important, it was only a part of a broader effort, an operating system; this should have been evident to the SCAT committee. However, the different opinions of a system of the different members of the committee –volunteers provided by the interested companies– made it challenging [Shell, 1959, 123].

Nevertheless, the committee was committed to specifying a "complete installation operational system with all the fundamental necessities for effective machine utilization" [Shell, 1959, 124]. In assessing the effort required to obtain such a system, the committee decided to leave the implementation to IBM, reiterating the strong ties between SHARE as a user group and the manufacturer [Shell, 1959, 125]. In the end, the 709 System or SHARE Operating System launched a complete but fragmentary solution intended to provide the programmer and the machine operator a "highly flexible complex of languages, procedures and machine codes" [Homan and Swindle, 1959, 4]. This description once again manifests the relevance of language, which becomes apparent when reading the SOS reference manual. Here, the SOS is presented as an integrated system composed of the SCAT subsystem, a debugging system, an input/output system, and a monitor [International Business Machines Corporation, 1960, 29,30], with SCAT representing a large part of the document. Although the development of the Operating System was a specification made by SHARE as a group (the 709 system committee) and the implementation headed by IBM, the process was a coordinated effort between SHARE and the manufacturer. The first phase of the work lasted eighteen months; the latter was brimming with "numerous modifications and clarifications" by the parties [International Business Machines Corporation, 1960, 11]. Ultimately, the 709 Operating System, optimistically called the SHARE Operating System (SOS), was ill-fated; IBM withdrew it in favor of alternatives developed entirely in-house. The system remains an example of a collaborative effort that became tangible. It is also an important and seminal project in which discussions and decisions were carried out collectively and cooperatively. The SOS's value is glaring considering its early development, in a time when topics such as programming languages, systems administration, and others were still acquiring their momentum.

To sum up, the SHARE user group can be regarded as a pioneering collaborative undertaking in which unformalized software development practices simply occurred, driven by significant pragmatism, when computing was still

in the making. Some additional points can be made to illustrate its relevance further. First, the contested linearity of some historical computing accounts is again challenged when looking at the first stage of SHARE (roughly from its foundation in 1955 to 1960). Modern FLOSS narratives consider IBM's adoption of Linux in early 2000 a milestone[11], whose significance –though important– changes when the time frame is broadened to encompass the motives, objectives, and operation modes of an organization such as SHARE in the late 1950s[12]. Secondly, materiality is very present through language and communications. For example, think of the relevance of face-to-face meetings at a time when computer networks sharing a common symbolical language did not exist; this was materialized by the "SQUOZE deck" (deck of punched cards) [Boehm and Steel, 1959, 136] and SCAT generally implementing "systems" and "programming macros" [Greenwald and Martin, 1956, 131]. These macros reveal both a symbolic division of labor (programming and system administration were already perceived differently) and the incremental and cumulative nature of language (macros are composites of a commonly used sequence of more straightforward commands) applied to computing. Thirdly, from a rather loose cybernetic perspective, it can be argued that the SHARE community was aware of the cybernetic metaphor[13] applied to the interaction between the user group and IBM (represented by the hardware improvements made by the company based on SHARE recommendations) as suggested by "the feedback" –a term that was not so fashionable at the time, therefore, highlighted with curiosity – mentioned in the accounts of members of the group [Armer, 1980, 127].

SHARE's cooperative effort has been praised as a truly cooperative and collaborative enterprise, as noted by the words "cooperative spirit" that worked at an age when the concept of the stored-program was infusing life into the future software culture. Perhaps, as a testimony of the SHARE user group's special place in the history of computing and software at large, we can cite the principles to which SHARE was committed. In a 1956 study by F. Jones entitled, "SHARE-A Study in the Reduction of Redundant Program-

[11] Coming sections will further discuss this point. For now, I mention the account from Glenn Moody's "Rebel code".

[12] SHARE still exists today and is dedicated to software exchange and providing feedback to IBM. However, it is considered a standard user group divested of the relevance of its founding years. See `http://www.share.org`

[13] I am aware of the breadth of *cybernetics*. However, I am referring here to the shift from first to second-order cybernetics, specifically, when the inclusion of the observer in the system allowed the metaphor to be extended to organizations. This change can also be suitably placed in the 50s [Hayles, 1999, 10].

ming Effort through the Promotion of Inter-Installation Communication", he classifies them as "1) the standardization of machine language and certain machine practices, 2) the elimination of redundant effort expended in connection with use of the computer, 3) the promotion of inter-installation communication, and 4) the development of a meaningful stream of information between the user and the manufacturer" [Guendert, 2011, 1]. One may even be tempted to compare this wording –at least the third and fourth points– with the so-called four liberties of the free software movement, coined almost three decades later in a movement to emphasize the community component of collaboration in software development and the traceability –although not linear– of some of its tropes. Ultimately, SHARE stands out as an influence that motivated other manufacturers to promote their own user groups, such as USE for UNIVAC or DECUS for DEC. It also left a legacy that is "the demonstration that cooperative undertakings by groups with diversified interests can succeed and can speed up the development of the art of machine computation" [Melahn, 1956, 269].

3.3 Interactive dreams: A hacker's tale

The turn of the decade (from the 1950s to the 1960s) marked several changes in the development of tandem computer software. The significance of these changes shaped the field with a series of advances that brought its practices closer to the modern conception of computing. Above all, they provided a new socio-technical environment where open, collaborative practices in software development acquired a broader scope, reaching in the so-called hacker culture a momentum that still feeds a large part of current socio-technical discourses, ranging from FLOSS to political activism. Thus, the changes considered create a tangle that is reissued through concerns about access and craftsmanship –the material and social condition of software. First, we must consider the changes in hardware. If the 1950s was the decade of the vacuum tube computer (the IBM 709 –the computer of choice for the SHARE Operating System– was the company's last vacuum tube mainframe), the 1960s heralded the full arrival of the transistor as the quintessential discrete element in computer architecture. Second, and related to the advent of transistor-based computers, was the development of real-time interactive computing. Although this feature was already achieved with vacuum tube technology, notably, in the legendary M.I.T Whirlwind (1951) and the subsequent SAGE system [Campbell-Kelly, 2004, 37], its impact did not reach the growing computer industry, and it was overlooked in favor of the batch approach. Third, was the birth of the idea of networking as a way to con-

nect computers. Though this advance was implemented at the end of the decade (discussed further in the next chapter), its theoretical principles and experiments took place during this time. Its development and history are intertwined with the narrative pursued here.

Before continuing, a brief digression is needed to evaluate the radical change represented by the paradigm shift in real-time interactivity. From the perspective of language, the linearity of programming became fragmented due to the undeniable unpredictability of interactive computing. Although some advances in computer languages paved the way to this, particularly branch and jump instructions, the dominant batch process approach relegated the computer to a more passive role, where its processes (and I will argue programming) obeyed a linear pattern. The machine waited for input, computed, and ultimately produced a print-out with results[14]. The problem with this representation of computing, namely, batch processing, is that it reinforces an instrumental perception of programming in which problem-solving was simply a faster method to calculate, an expedited means to an end. In other words, it portrays the computer as a *better calculator machine*, not as a *brand new machine*, backed by the flexibility of the affordances of software.

To support the former claim, a reading of the effects of this real-time interactivity from a cybernetic register can point to two aspects, which are crucial in understanding the crossroads between software as a technology, collaboration, and the community, namely, craftiness and accessibility. On the one hand, the transistor's interactivity allowed the human-machine circuit to respond at a more human-like speed, granting the software development tasks new qualities. Thus, the programmer was able to edit, compile, examine, and debug a program without the hassles of batch processing. This made software more hands-on, granting software developers more agency to control the flow of their code, directly associated with the technological tradition of tinkering. Currently, several accounts commonly consider software development as a kind of art or craft based on iterations in trial-error approaches[15]. On the other hand, and considering the socio-technical system surrounding computers, the appearance of the transistor-based minicomputer

[14]This trend is, to my understanding, rooted in a rationalistic computing perspective that manifests before the advent of the electronic computer in calculations by human computer efforts or punch card equipment.

[15]I am not trying to draw an opposing distinction between a rationally driven type of batch processing versus intuitive tinkering interactivity. However, it is essential acknowledge that in a batch processing environments where access to the computer was scarce, more planning was required to use the available resources more efficiently.

unleashed a new range of possibilities in which the interactive computing approach questioned the common access barriers of the batch model (while this model did not disappear). Compared to the expensive mainframe computers used, for example, in the SHARE efforts, the more affordable minicomputers facilitated new forms of computer interaction, which would lead to new developments in the socio-technical system where the computer, software, and community intertwine. Although computers were by no means as common as they are today, and communities of professionals had limited access to these resources, and even though transistor technology was also used in mainframes, the minicomputer was a different kind of machine with another paradigm of achieving things that encouraged a hands-on approach. This relationship is paramount to understanding the emergence of a community that, in many ways, catalyzed computing as a practice, and whose tropes still inform the modern conception of open collaboration practices in software. I am referring to the hacker community, or in general, the so-called "hacker culture".

Characterizing the hacker culture is not an easy task, considering space constraints and the fact that it is profoundly heterogeneous [Coleman, 2012]. However, to do so, I consider it necessary to establish the premises on which I will base the brief account offered. First, if we accept the existence of software culture, we must acknowledge that the hacker culture is one of its essential components. The relationship becomes vital when considering that the figure of the hacker has become paramount in describing the current discourses on open collaboration software practices and how these involve a metaphor that is used in various disciplines. Secondly, as suggested previously, the hacker culture is not without its tensions, as can be observed when evaluating the so-called hacker ethos and its acceptance by divergent positions (e.g., the discourse of Silicon Valley capitalist entrepreneurship and various global activist groups). Lastly, this culture has mutated over time, making it challenging to frame, and the narratives that try to describe it have several problems. Therefore, I have divided the hacker culture sequence into three instances (or *iterations*). The first iteration, the allegedly original MIT hacker culture, is discussed in this section (the other two iterations are the hacker culture within the FLOSS movement and the network culture). To illustrate how this hacker culture was set into motion, the incompatibility time-sharing system (ITS) and the UNIX operating systems are also discussed. These two examples are useful to understand more clearly the hacker ethos at the core of the hacker culture and its interface with real collaborative software projects. While this historical approach has a critical focus, the latter is not intended to diminish the importance of the hacker

culture, but rather to discuss some of the controversies that have arisen from viewing the hacker culture as a concept of linear evolution.

3.3.1 First iteration: the hacker culture

As mentioned, the hacker culture is an integral part of the narrative of open collaboration in software and the software culture at large. The popularity that the terms hacker or hacking have attained reveals, at the same time, the fascination with hacker tropes and the flexibility of the hacker discourse. If open collaboration in software production can be seen as a metaphor that mutates and translates into other contexts, the figure of the hacker and the hacker community's set of practices underwent a similar process[16]. As a unit, the study of the hacker culture has a canonical reference work dating from the 80s. The well-known book by Stephen Levy [Levy, 1994] is allegedly the first attempt on the subject and perhaps, the most referenced. Although Levy's account is journalistic and lacks an academic setting, the richness of the interviews with first-generation hackers and his effort to distill the hacker culture's main tropes makes it his main contribution. Following the trend of scholarly dialog, with Levy's work intending to broaden and frame it, I contrast it with other sources to briefly address some of the elements of the hacker culture that are related to its particular exercise of technology and its influence on open collaboration software patterns. It should be noted that, despite its current prevalence, this first iteration on the hacker culture considers a particular geographical and temporal context. The famous MIT from the late '50s through the late '60s is purportedly the *cradle* of the hacker culture.

From a technological perspective, the hacker culture and its process represent a modern development on the relevance of the concept (technology). Its status is a motor of society, entangled with human nature and, at the same time, not conditioned by science. The hacker community's celebrated hands-on approach to computer programming, their feverish trial and error problem-solving method using inventive means seem detached from the sys-

[16]Once again, I emphasize the concept of *community* related to a set of values and practices embodied in a culture. However, I also want to avoid, for now, the discussion on gender issues within the hacker culture. There is a marked contradiction in assessing the human factor of hacker practices when technologies, interests, and even political positions are taken into account; this community is diverse. However, when demographics are considered, the hacker community is seen to be composed mainly of white techno-savvy males. Therefore, while perhaps the hacker (he) could be more specific, this tension is left to the end.

tematic step-by-step rigor commonly attributed to science. As discussed in the Second Chapter, the tension between science and technology, the latter emerging as an entirely independent and valuable field of research, regards technology as a more intuitive and less rational effort, closer to arts than to physics. By no means does this interpretation disregard the influence of rationalism on the emergence of computing or in the apparent notation related to the symbolic substrate of software. Instead, it is an acknowledgment of the fact that the flexibility provided by software, perceived as a technology, escapes the already contested discourse of technology as an instrument dominated by scientific supremacy.

Moreover, from the onset of the so-called crisis of software, which somewhat coincides with the origin of computer science and software engineering as disciplines, had to address the issue of programming as more than just a calculation activity, rather, an activity that also allowed non systematic approaches . Computing is determined according to some axioms when "it becomes software (...) come into combination with what lies outside code"[17] [Fuller, 2008, 5]. As a side note, Levy separates hackers –obsessed with computation per se– from the group of planners, usually composed of scientists focused on the results of computer data processing to solve a scientific problem that were calling computing a science [Levy, 1994, 57,58]. However, the latter does not imply an opposition of the hacker-planners duple or technology-science for that matter. Some of the most memorable achievements of the hacker method can be found in projects outlined originally by planners (the LISP language or the ITS operating system as notable examples). However, the work ethics of hackers cannot be explained by normal standards, and, as noted by Levy, its participants cannot be managed; instead, it suggests a technical problem for the hacker's curiosity to address [Levy, 1994, 115].

At this point, it should be clear that the supposedly original MIT hacker culture (and its subsequent variations) cannot be correctly described using a merely scientific setting and that the hacker production mode is closer to the engineering approach. Therefore, this observation alone cannot explain the emergence of open, collaborative practices related to software production in the hacker culture. Hence, I point to the symbiotic relationship between craft and pragmatics and the tensions that lie in the middle to address this

[17]My interpretation of this quote is twofold; it shows software as more than a rational artifact defined by a pure axiom and establishes the importance of the context where software is created, hence the *culture* that surrounds it.

issue. First, the association of the hacker figure with an artisan, whose work (in this case computer code) displays features of craftiness, is evident among the mix of anecdotal and scholar readings on the hacker culture. For instance, in Graham's view, programming is a type of craft, like painting, but definitively not a science [Graham, 2010, 18,93]. Richard Sennett, regarding the Linux community, considers the key features of control, the freedom to experiment, and inner satisfaction as appropriate examples in his quote of C. Wright Mills on the character of the craftsman [Sennett, 2008, 27]. Levy himself, when talking about the passion of a particular programmer and his skills, wrote that hacking "was above all a pride in craftsmanship" [Levy, 1994, 92]. Perhaps, a statement in the documentary, "Hackers: Wizards of the Computer Age", (although dating back almost three decades after MIT's first wave of the hacker culture) can sum up the interaction at play in the hacker's programming approach. In this documentary, an interviewee asserts that the "systematic approach to software development misses the boat" and that "trial and error" was the proper method to approach it, of course, as done by the hackers[18] [Florin, 1986].

To continue a previous discussion, I believe that the emergence of interactive computing is fundamental to the development of this hacker-artisan persona, essential to the hacker culture narrative. If an intuitive reading of craftiness highlights the importance of the hands (and hand-dimensioned tools) in the artisan's activity, then, the affordances of interactive computing and the possibility of accessing the machine directly and in real-time offered a similar faculty to programmers during the period (late '50s and '60s) when this technology was introduced. In accepting this claim, the common etymological root between the craftsman and *hand-made*, and the hacker and the *hands-on* tinkering method, could be assumed from a more significant perspective.

Software's dependency on hardware is again re-enacted through the connection of this original hacker culture and the DEC interactive mini-computer platform. This platform was the quintessential hacker computer and whose lineage became the most appropriate choice for universities and research institutes, including MIT [Campbell-Kelly and Aspray, 1996, 225]. In fact, hacker culture critics such as Raymond, trace the origin of this culture back to 1961, when the MIT acquired its first PDP-1 interactive machine [Raymond, 1999, 16]. Having established hacking's craft quality and the quest

[18] As stated before, these references are journalistic accounts. However, they are valuable in providing an overview on the main tropes of the hacker culture.

for excellence associated with it, the shift to open collaboration was easily achieved with the help of pragmatism. That is, the ability to exchange and modify computer code was a condition for obtaining more elegant and well-achieved programs, in short, beautiful software. Pragmatism was also a force behind experiences like the SHARE computer users group, but what was different? While this observation holds some truth, several key differences can be found between the corporate batch processing paradigm and the new interactive hacker mode. As a project descended from corporations using IBM equipment, SHARE pragmatism was well-rooted in collaboration as a tool to reduce high software development costs through program sharing, coordinated efforts in collective endeavors, and the introduction of standards from hardware configurations to layers of abstraction (e.g., compilers). Besides having financial motivations, the SHARE hierarchical structure reflects its corporate nature. The hacker culture, on the other hand, emerged in an academic environment, where the search for profit (at least at the time) was not the engine of innovation. Several other descriptors have been discussed when assessing the hacker's motivation, desire for recognition and control, as well as pride, craftsmanship, and fun, are among the most common. In terms of its approach, the hacker culture also diverges from previous batch-oriented recommendations. It displays a *laissez-faire* attitude that allowed for productivity [Levy, 1994, 115] when the mere curiosity of controlling a new piece of equipment (such as a robot arm) could spark the hacker's drive [Levy, 1994, 64]. It can certainly be added that the hacker culture's distinctive feature of *productive freedom* "designates the institutions, legal devices and moral codes that hackers have built in order to autonomously improve on their peers' work, refine their technical skills, and extend craft like engineering traditions" [Coleman, 2012, 3]. In other words, the hacker culture itself is a construction that fosters the flow of code and technical excellence under a pragmatic imperative. Ultimately, while the batch culture and the hacker culture have coexisted together, it was the latter that became a cultural referent outside of computing, and the reason is the seductive yet problematic "hacker ethic".

Steven Levy's enunciation of the hacker ethic in his hackers account, which has endured as the main reference to the topic since it was first published in 1984, instructs that (paraphrasing) access to computers should be unlimited and the hands-on imperative should be yielded, all information should be free, distrust authority, and promote decentralization, hackers should be judged only by their skills, art can be created with the computer, and computers can change lives for the better [Levy, 1994, 28-34]. It should be noted that in assessing this narrative as a whole, the solely pragmatic reason is

replaced by a broader imaginary where art and a social concern could be included. In doing so, the figure of the artisan is amplified with an aesthetic and political agency that shapes an irresistible character, indeed framed by the IT revival of the alluring romantic aura and its correspondent utopias [Coyne, 1999, 10]. Commonly, the philosophy of computing is full of gaps, and innovation in the field raises new ethical concerns [Bork, 2008, 2]; the ethics of the hacker is undoubtedly an excellent example of this and is not free of contradictions. The success of the concept propelled other similar inquiries. For example, the self-titled "Hacker Ethic", which directly declares it the *"new work ethic"* (his emphasis), challenges previous treatments of the topic [Himanen, 1999, ix].

The manifesto-style hacker ethics, written by Levy, are a posterior systematization effort by the author based on his interviews and not by assertions of the hackers themselves. This in no way invalidates the way these early hackers are depicted; instead, it articulates the problem of self-awareness of this first iteration of the hacker culture. For example, in describing the background of the hacker culture, Raymond points to the culture of the real programmers, as a tradition dating from 1945 onwards that "was a more or less continuous and self-conscious technical culture of enthusiast programmers, people who built and played with software for fun" [Raymond, 1999, 16]. Although this description does not exactly belong to the hacker culture, this culture's condition as its precursor, as well as its technical nature and its motivation of fun establish a continuity. In a contrary but historical observation, through field studies among newer incarnations of the hacker cultures, Coleman differentiates between an original and oblivious hacker culture and a self-aware one. The latter exemplified by the emergence of the free software movement and the GNU Project and how its actors put their reading of the hacker ideology on paper [Coleman, 2012, 18]. In the previously mentioned documentary on hackers, an interviewee cheerfully admits that he was aware of the MIT hacker culture, only when he is far from it [Florin, 1986]. Ultimately, the hacker culture contributes to the understanding of the open collaboration practices in the more general software culture because much of the discursive arguments on these practices are largely indebted to how the hacker narrative has been embedded in them.

Moreover, it is hacker ethic, which can be read from a simple pragmatic or a more altruistic perspective, that offers a heterogeneous field (despite the general perception) whose tensions have continually resurfaced in the contemporary incarnations of the hacker culture. Likewise, it provides a convenient metaphor that can be extended because, as expressed in hacker

jargon, a hacker can be "an expert or enthusiast of any kind" [The Jargon File (version 4.4.7), nd]; this connects directly to amateur culture and joyful motivations. However, the hacker culture can also be perceived as a demonstration of how "autonomous, self-generated circuits of prestige of the gift economy produce self-generated circuits of extraordinary innovation"[19] [Wark, 2004, 071].

3.3.2 Shared resources: the ITS

To broaden the understanding of the hacker culture, its ethics, and the importance of its ideas in shaping the modern idea of open collaboration practices from the historical framing of this chapter, the discussion of one of this culture's products can be useful. The ITS (Incompatible Time-Sharing System) operative system is once again mentioned not only because of its preponderance in the hacker saga but also in the context surrounding it in which the interaction between language, materiality, and craftiness was perceptible. The ITS also represents a pivotal point in systems development; its introduction reflects the relevance of an open environment and a step further into the distributed networked concept of software collaboration. On the one hand, and continuing with the early history of modern MIT hackerdom, the ITS was developed at a laboratory where non-traditional management and tinkering were at the service of technological innovation. On the other, the ITS is an implementation of the time-sharing architecture of an operating system, a first theoretical, then, pragmatic advancement of computing systems programming in the late '50s and through the '60s. Both the open laboratory model and the time-sharing system are entwined with the role of a community in a socio-technical system, some of their suggestions would later populate the rhetoric and methodology of open collaboration software and its consequences.

The ITS operating system was a product of the MIT hacker culture. It foresaw some of the collaborative possibilities that became feasible some years later with the emergence of computer networks. Specifically, the ITS, which was developed within the MAC project, was an MIT effort to obtain military funding from the fledgling ARPA (Advanced Research Projects Agency). The ARPA was created by the government of the United States to foster technological research amidst the technological progress race that characterized the cold war. This project was developed to bypass the bureaucratic

[19]This was a comment by Wark on the benefits of the gift economy as described in the hackers account by Levy.

inconveniences of being called "laboratory" and to be able to recruit personal from different departments, allowing them "to maintain their existing laboratory affiliations" [Chiou et al., 2001, 12]. This maneuver proved its value in assessing the heterogeneity of the laboratory as a *social construct* in which different interests associated with the still-maturing field of computing; it encompassed influential laboratories in Artificial Intelligence and Computer Science. The name MAC, which initially stood for "machine-aided cognition" and "multiple access computer" [Levy, 1994, 59], epitomized the research aims of the laboratory, where theoretical (as in cognition) and pragmatic interests could be negotiated and pursued.

Clearly, the role of the multiple access computer connects directly with the time-sharing endeavors that marked the time. It provided a conceptual framework to explore the cybernetic quest for suitable man-machine interaction under the interactive computing imperative associated with hacker ethics. The managerial style of the MAC project is described as quite relaxed. It built an interface between the typical university bureaucracy (although a government-funded project) and the hacker approach to systems development that operated outside standard plans and supervision [Levy, 1994, 59]. As one of the heads of the project, A.I. pioneer Marvin Minsky delegated his responsibilities and focused on attracting a programming force by encouraging *hackerism*, fostering an open environment where "everybody who wanted to play could come play with his toys", meaning costly A.I. equipment and computers [Chiou et al., 2001, 8]. The MAC project managerial factor is noteworthy because it displays the "open community rationale" so often mentioned when evaluating collaboration in software.

Additionally, given the historical context where networks did not still adequately exist, time-sharing points to a middle ground where physicality was still necessary to collaborate in a software project (although remote terminals were possible). This peer-to-peer/face-to-face dichotomy would inform some future software development methodologies and applications outside the realm of software development. However, it remained a product of its influence, which was reflected in the evident promotion of the workshop by the maker culture as a collaborative space. Once again, the *community* must be acknowledged as a fundamental part of the socio-technical arrangement of software. In this direction, the MAC project had, from the beginning, the purpose of assembling a community composed of computer researchers from different departments [Chiou et al., 2001, 5]. Ultimately, the MAC project supplied not only a kind of canonical example of early hacker culture (and its problems within an organization) but, at large, "explored the potential of

communities on time-sharing machines" [Salus, 2008, 6].

As previously discussed, time-sharing systems were an important research area at the end of the '50s and early '60s. The increasing cost of computation (particularly software development) represented a constant driver of research at a time when the batch processing paradigm was beginning to show signs of exhaustion in certain fields. Even though batch processing met the requirements of classic business-oriented applications such as payroll, other applications began to demand interactive human-machine procedures. The combination of the cost-effective use of computer resources use, and new interactive uses of the machine led to the emergence of the time-sharing concept in which this costly machine called computer – which ineffectively could be used for one process at a time – allowed several processes and users simultaneously. Another MIT associated name comes forth when examining the concept of time-sharing, A.I. pioneer John McCarthy. With Minsky, McCarthy co-directed the beginnings of the MAC project before going to Stanford. Apparently, it was McCarthy who first thought of time-sharing around 1955 (or at least the definition that concerns me). In 1959, he wrote a famous memorandum advocating the concept as a solution to the shortage of computer resources at MIT[20]. McCarthy's recommendations are a portrayal of the relevance of hardware to the advent of time-sharing and software in general. He was very aware that time-sharing would not be feasible with the lab's IBM 704 (the same of the SHARE user group) without a hardware modification, which he proposed in 1957 [Salus, 2008, 6] citing [McCarthy, 1992, 19]. Years later, this modification was implemented in an IBM 709 to provide a "trapping mode" to the CTSS, a predecessor of the ITS [Chiou et al., 2001, 14].

The ITS was also dependent on the features of the computing model offered by the PDP manufacturer (a company with strong bonds with the early hacker culture), which resulted from the interactivity allowed by the shift to a transistor-based architecture[21]. As a result, time-sharing can be seen as a combination of economic motivations and both new research challenges and programming requirements, especially debugging. As McCarthy summed up, "[t]he only way quick response can be provided at a bearable cost is by time-sharing" [McCarthy, 1959]. It should be explained here that

[20]Perhaps it would be more fair to say that *the concept was in the air* and McCarthy put the parts together [Lee, 1992, 16] and [Lee et al., 1992, 18].

[21]Reasonably enough, if the image of vacuum tubes computers are usually associated to the batch processing paradigm, then transistors could be related to the new advancement of time-sharing.

"time-sharing" had two meanings at that time. The first one pertained to different computer operations being executed simultaneously; that is, when a task had to wait for input/output to switch to another task, a feature also known as multiprogramming. The second was related to multiple users using the computer interactively [Lee, 1992, 17]. Although the first definition entails the achievement of the second, it is the latter that has major relevance to the argument of open collaboration in software[22].

Combining the previous points, ITS represents an operating system that offered revolutionary features from a collaborative software development perspective and reflects the thread of the hacker culture embedded in some modern open collaboration tropes. More specifically, ITS was an answer to the increasing demand for interactive debugging in software development. Batch processing lengthened the entire development process by preventing programmer-computer interaction [Chiou et al., 2001, 1]. As the seminal place occupied by the MAC project would demonstrate, interactive debugging would prove valuable to the revolutionary computing advances (in the systems and IA fields) being made there. ITS not only offered the interactive debugging favored by McCarthy instead of the painstaking core-dump readings but it also implemented an open architecture where all users could read and write each other's files [Eastlake, 1972, 5]. They were actually encouraged "to see what new projects they [other users] were working on, find bugs, and fix them" [Chiou et al., 2001, 28]. Furthermore, users could switch to other users' displays and join the program effort; this can be considered as a precursor of practices such as pair-programming [Chiou et al., 2001, 28]. Technically speaking, ITS was the realization of the hacker's hands-on imperative applied to interactive and collaborative program development through time-sharing. It was a system that drove openness to the extreme, considering that it did not implement any password security protocols, something seen as coherent in a "comprehensive open research environment" such as project MAC [Eastlake, 1972, 5].

Despite the relevance of ITS in the history of open collaboration in software, certain aspects must be discussed to establish a critical viewpoint that will

[22]This adds further confusion to the origin of the term time-sharing. Following an analysis of the topic, the multiprogramming reading of the term was in usage before 1960. After that came the interactive interpretation (from the staff of the MAC project) [Lee, 1992, 17] and [Corbató et al., 1962, 335]. McCarthy explains that the SAGE project before had something similar to time-sharing and that the Strachey's (credited sometimes for the time-sharing concept) definition does not refer to multiple users, a statement that Strachey confirmed [McCarthy, 1992, 23].

improve the evaluation of this operating system. From a language analysis perspective, for example, time-sharing furthered the breakdown of instruction processing linearity. As McCarthy's memorandum states, in the '50s, regular computer use was quite the opposite of the original conception of an electronic calculator, an approach represented by batch processing [McCarthy, 1959]. As already discussed, interactivity represented a pragmatic opposition to linear processing; an effect increased when considered in conjunction with time-sharing (several users, each entitled to an interactive terminal). Branching and jumping were not sufficient to achieve the time-sharing of a machine. Therefore, new programming and hardware mechanisms (such as memory protection and segmentation) were proposed to support this new man-machine communication paradigm [Corbató et al., 1962, 335-337]. From a hardware language perspective, the decision by ITS developers to stick to the PDP-11 machine language provided the performance and reliability envisioned by the hackers to support collaborative development activities [Levy, 1994, 118]. At the same time, it condemned the operative system to be a non-portable creature, which never had extensive use outside the MIT research community. In the '70s, it was also dropped when it was considered for the early Arpanet because of its lack of security [Chiou et al., 2001, 39,40].

As suggested, the inability of ITS to transcend points to other inconsistencies in the representations of the hacker culture depictions in which it was supposedly embedded. Time-sharing, for example, was a kind of anathema for hackers, who weighted it as a value against their computing priority of qualitative interactivity, in fact, the MAC project management had to convince them to take part in this burgeoning field of system research [Levy, 1994, 116,117]. Despite the success of the system, hackers were not interested in disseminating its benefits beyond their ideal "Xanadu" of code [Levy, 1994, 123] let alone trying to change the mainstream corporate batch-oriented computing culture; interestingly, the precursor to the ITS, the CTSS (Compatible Time-Sharing System), originally had this objective [Chiou et al., 2001, 31]. However, from this workforce perspective, the ITS can be fully appreciated when compared to another two time-sharing systems being produced in the MAC project, the mentioned CTSS, and Multics (Multiplexed Information and Computing Service). Both projects aimed at similar collaboration tropes. The CTSS planned to bring the "computer to the masses", and the Multics design included several (then radical) communications tools and useful aids to do so [Multicians, 2015]. However, both systems were outlined as hierarchical processes (actually belonging to the same managerial culture), including decision-making committees and a division of labor between design and implementation [Chiou et al., 2001, 18]. On the other side of the spectrum, the

ITS was gradually developed by small groups of two or three persons –designers and coders. Thanks to this double facet, this decision embodied the philosophy of a less risky software process [Eastlake, 1972, 4]. Ultimately, the ITS stands as an early experiment in open collaboration applied to software development that benefited from an open environment (MAC project), from practices that anticipated trends in software engineering, and from a system that, in turn, adopted code-level collaboration. Similarly, the longevity of the system at MIT (from 1967 to 1990), which in the end was more of an exercise in nostalgia, demonstrates its technical excellence and its relevant status as a testimony of a culture in the making (the hacker culture), where hands-on approaches, free flow of information, and innovation articulated an effective, though not always controversy-free, process of software production.

3.3.3 The UNIX philosophy

Although the last section presented the ITS as an operating system that embodies open collaboration practices, in processes and product features, as interpreted by the ethics of the early and still influential hackers, in perspective, it is just another undertaking among many in the developing field of time-sharing systems at the time. Of all the systems that were developed in the early '60s, the two already mentioned (also developed in project MAC), the CTSS and Multics, are relevant to build a discursive context where collaboration interactions can be addressed from a socio-technical system perspective. The CTSS was an effort initiated by Fernando Corbató in the MAC project (circa 1962); it was an early assessment of the possibility of time-sharing. The CTSS took a rather conservative approach by allowing several users to operate interactively, side by side with the regular batch-oriented operating system (hence, *compatible*), by "stealing" time from processing [Chiou et al., 2001, 13]. On the other hand, Multics, which started in 1964, was a massive undertaking that involved tripartite management by the three members of the enterprise (MIT, General Electric, and AT&T Bell Telephone Laboratories), which implied a significant managerial effort. Despite its revolutionary design, which produced innovations that are now taken for granted (process communication and a hierarchical file system), it is remembered as a great promise never fulfilled, a failure. Two main differences between the CTSS and Multics when dealing with ITS are, as noted, the strict division of labor (of designers and coders) and coordination through committees. The difference of the first two with the somewhat chaotic and voluntary approach embodied by the hacker ethics of the latter is a distinction that is deepened by the radical collaborative ethos embedded in the ITS design.

However, and although tempting, it would not be entirely accurate to establish an opposition between these operating systems, given their common roots in the MAC project, as well as the fact that neither achieved widespread use. Multics –although seminal– never reached a fully functional state, and both the CTSS and ITS never left the vicinity of MIT. The drawback of establishing an opposing framework is also supported when considering that one of the objectives was the dissemination of time-sharing computing, a goal present in the "computer as a utility" rationale of CTSS and Multics, and yet somehow contradictorily not so openly embraced by the hackers behind ITS, despite what one can infer from the phrasing of the hacker ethics[23]. In contrast, the novel idea of an operating system as a research topic nurtured several innovations. It provided a testing ground for the hacker ethics and their hands-on (matched with craftsmanship) mode of production. Consequently and despite the appeal and relevance of the hacker narrative in open collaboration practices in software production or the relevance of Multics as a model, the discussion would be pointless without a successful example. This ambition is fulfilled by the Unix operating system, a synthesis of several previous practices that was effectively and widely adopted.

The history of Unix is a perfect example of a software product that is, from a pragmatic point of view, a real system whose successors (look-alike systems, forked systems, and mobile systems) consider the most widespread operating system. Similarly, it is a system that, from its origins, displayed the tension between hackerism and corporate management and also, a system whose evolution relied heavily on an organic community with open collaboration mechanisms at a crucial time when computer networks were emerging[24].

Work on the Unix operating system began in 1969, mainly by Ken Thomson Dennis Ritchie at AT&T Bell Telephone Laboratories (BTL). Although their involvement is sometimes overlooked, J.F. Ossana and R. Morris were also

[23]It is worth briefly pointing out the issues of the notion of computer utility. This computer utility approach intended to provide researchers from different areas tools to do their work. Although this utilitarian view of computing holds in various disciplines, it offers an incomplete framework of the multidimensionality of computing when considering software practices. The difference between this idea an implementation like ITS is that, in the latter, the operating system was the object of inquiry through incremental development rather than just a tool for other purposes. This, in turn, can be read from the originary technicity substrate of software that escapes instrumentality to the tension between modes of software production at the dawn of software engineering

[24]From the long saga of the Unix operating system, I will select only some data, useful to my purpose. Exhaustive accounts of this history can be found in various sources, particularly in the work of Unix historian Paul H. Salus [Salus, 1994].

involved in the origin of the system. Ritchie, dissatisfied with the end of the Multics project and the loss of the computing services it provided (although never completed), decided not to stand idly by as AT&T stepped away from the project [Ritchie, 1984, 1577]. It is important to note that Unix was not the result of managerial planning like Multics; it was a personal effort driven by the dissatisfaction produced by the absence of the powerful Multics operating system and its features [Ritchie, 1984, 1578]. As a matter of fact, the file system, one of the first Unix usable parts –which encouraged further development when completed– was programmed by Thomson in a month when his wife was away [Toomey, 2011, 36]. This early development style based on *hacking away* connects the beginnings of Unix directly with ITS and its hacker culture nature. It also stands as an example of innovation without clear planning and following a tinkering approach. According to Thompson (1974), "the success of UNIX is largely due to the fact that it was not designed to meet any predefined objectives" [Ritchie and Thompson, 1974, 373]. The then developing system filled the void left by Multics[25].

By 1971, when it (Unix) was operational, it had reached a user base of about forty installations [Ritchie and Thompson, 1974, 365]. Its ability to be adopted by institutions other than Bell Laboratories was thanks to the simplicity of the system, which could be run on inexpensive hardware and was developed in "less than two-man years" [Ritchie and Thompson, 1974, 365]. This ethos of simplicity not only marks a departure from the course attempted by Multics (bureaucracy and grandiose design) but in a similar way to the hacker ethics, it was expressed later in what is now known as the "Unix philosophy". However, simplicity alone cannot explain the enormous presence of Unix in contemporary computing and systems design; for that, we have to look at the role of the community in a socio-technical system and the open protocols created by its members. In this sense, Unix prolonged one of Multics' and time-sharing in general's objectives as explained in this retrospective by Ritchie "[w]hat we wanted to preserve was not a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication" [Ritchie, 1984, 1578]. In other words, the programmer's pragmatism was reinforced by a collective aspiration that ex-

[25]According to folklore the name Unix comes from Unics, which was a play on Multics, given that Unics was inspired on the larger system but only allowed a single user at a time.

plicitly represented the legacy of Multics, a system that, while not complete, was a good design tool and fostered a fellowship [Toomey, 2011, 36]. This prospect of a collaborative environment gained momentum when Unix was finally presented to a broader audience in an operating systems conference in 1974, where it was received with enthusiasm and requests for copies [Toomey, 2011, 52]. Quite reasonably, the success of the operating system was closely related to the ability to access the code; this resulted in a thriving community interested not only in using Unix but also in improving it, fixing bugs, writing new tools, and providing mutual assistance in user networks at a global scale [Toomey, 2011, 53].

As the case has been in other examples of early open software production such as the SHARE group or the ITS operating system, access to the source code seems to be a necessary condition to produce a software collaborating community. In the case of Unix, when considering code from the language perspective in the other sections, the observation carries some interesting points. From an anecdotal perspective, because Unix was not planned and the Multics fiasco discouraged further systems research at BTL, the early development of the system was disguised from administrative control as a word processing system, assigning resources to a new machine. In the end, a complete operating system (including word processing tools) was built. The BTL patent office was the first user outside the core group, validating the system's viability and the relevance of the whole enterprise [Salus, 1994, 37]. From a more critical perspective, the free flow of source code and the agency that attained it implied the already discusses interdependency between hardware, programming, and community building. To illustrate, in the beginning, Unix was not very different from ITS. It was a system developed through hacker-style coding bursts, linked to the hacker culture venerated PDP machines, first, an outdated PDP-7 used when there was no interest in systems research at BTL, then, the PDP-11/20 and PDP-11/45 thanks to the text processing maneuver. Thus, Unix was also linked to these machines' assembly language.

It can be argued here that the coding of the system in a machine-independent language, such as C (originated within Unix), in 1973 [Ritchie, 1993, 9] was the initial step that allowed the adoption of the system in heterogeneous hardware architectures and, therefore, stimulated the development of a community around the system. Naturally, this portability movement required a C compiler on different machines, as is usually the case before more ports can be created. However, efforts in this direction had already been attempted in nearby machines, in the process, creating the "modern libraries" of the language [Ritchie, 1993, 9]. In the already mentioned 1974 conference, a small

internal operating system, coded by machine-independent language was presented to the public. Collaboration followed as almost a logical consequence, driven by the interests of the community and access to code, "the new [C coded] system is not only much easier to understand and to modify but also includes many functional improvements, including multiprogramming and the ability to share reentrant code" [Ritchie and Thompson, 1974, 366]. At this point, a small departure complements the idea of code and its legibility. Although C language had a broader appeal than assembly language, it was still difficult to evaluate. Despite the volumes of the "Unix Programmers Guide" that AT&T distributed to Unix licensees,[26] it took a text like John Lions' "Code and Commentary" –considered perhaps as the most reproduced book in the history of computation at the time– to arouse the popularity of the system by further explaining the accessible source code through explanatory comments [Salus, 2008, 22,23], again, pointing to a more organic conception of software, where documentation plays a vital role. This entanglement of hardware (or hardware independence), code access, and community was then incorporated –so to speak– into the design of the Unix system and its surrounding social practices. As expressed by Ritchie, "much of the success of UNIX follows from the readability, modifiability, and portability of its software that in turn follows from its expression high-level languages" [Ritchie, 1984, 1592].

In closing, it could be said that the importance of Unix has multiple interpretations. From the historical account of this chapter, the introduction and early evolution of Unix took place at the juncture when the beginnings of networking, software engineering, and commercialization of software were uncertain. From a more human labor perspective, the particularities of the Unix origins recreate the conflict between *planners* and *hackers*, and their radically different approach to software development. Last but not least, from a technological standpoint, Unix provided a field for open experimentation in which various system techniques, communications protocols, and software development innovations were put into practice. To briefly illustrate this socio-technical assembly and its accompanying communication patterns, let us review the following accounts:

On the one hand, the materiality of software –or at least its support– and its relationship to code circulation was manifest in tape-based software improve-

[26] I have not addressed the complicated intellectual property issue surrounding Unix collaborative practices at its early state, because I see this discussion as more relevant to next section, crisis of software.

ment and distribution. While the first copies of Unix were freely distributed on tape, the medium was used in reverse as the users submitted tapes with new applications or bug fixes to the core group in order to have their content included in later releases [Toomey, 2011, 53]. On the other, the already mentioned hardware independence was supported by the system's standalone feature, that is, that no other system was needed for cross-compiling or porting as with Multics; all Unix developments were made on Unix itself. Here, the availability of code again made the difference, "since all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others"[27][Ritchie and Thompson, 1974, 374]. From the first edition of the "Unix Programmer's Manual" by Ken Thomson and Dennis Ritchie, the practice of assigning an owner to each program was introduced, including login for the personnel responsible for a piece of code [Salus, 1994, 40,41]. This kind of tradition has not only survived many years in the Unix or Unix-like systems, but it was also an early example of a kind of code tracking system that used Arpanet and the newly invented email.

In short, the relevance of Unix lies in the fact that its practices typify the communal emergence of development practices that would later be systematized by the open-source concept. In doing so, an important interpretation of software engineering was applied, displaying a socio-technical evolution that explains the convergence of coding patterns and the modularity developed in the language associated with Unix, the C language [Spinellis et al., 2016, 748,749]. Similarly, it is important to highlight the value of an organized community, as was the case of the independently founded USENIX group, in the distribution of software and knowledge, all key factors in the healthy evolution of an open system [Toomey, 2011, 53]. Perhaps the same community synthesized its practices in the so-called "Unix philosophy"; a pragmatic construction developed over time that promotes simplicity and modularity as an approach to system design and implementation. This philosophy states[28][Salus, 1994, 53]:

- Write programs that do one thing and do it well.
- Write programs that work together.

[27] *On-line* is understood as available from the interactive prompt, following the principles of a time-sharing system.

[28] These simple ideas highlight the standard good practices of software engineering, such as promoting cohesion and avoiding coupling. They point to the importance of text as a communication medium beyond code.

- Write programs that handle text streams because that is a universal interface.

3.4 The crisis of software

One of the most common narratives in the software industry is that the field is in crisis, or even worse, that it has been in an ongoing crisis from the beginning. This rhetoric, which has permeated both business and academia, appeared in the late 1960s. It describes "those recurring system development problems in which software development problems cause the entire system to be late, over budget, not responsive to the user and/or customer requirements, and difficult to use, maintain, and enhance" [Dorfman and Thayer, 1999, 1]. The evidence that drives these concerns points to the increasing general complexity of software manifested in the exponential growth of lines code and the implementation of new technologies, such as time-sharing. The result has been failed projects like the paradigmatic development of the IBM operating systems for the 360 family machines in the early '60s. The importance of this so-called crisis is that the phrasing and the imaginary embodied in it are closely related to the emergence of software engineering as a discipline. This has served as an early framework for conflicts between software production models, namely, a model oriented towards a strict planned division of labor as opposed to a more freestyle hacker model. From this point of view, I use the term software crisis as a metaphor to describe the shift of open, collaborative practices in software development from being a common practice to being problematic and questioned. This movement, which could be considered a deviation from the original meaning of the crisis, can best be appreciated as a development of the original concept that explains this particular period in which the software industry exploded. The discipline of the software engineering tried to bring order to a chaotic field. The former premises of collaboration as shared code, were put into question by software commodification.

From this point of view, the expanded software crisis involves three main issues, which together provide a better understanding of a more modern conception of software and its development. The following sections describe these issues. The first section highlights software engineering's concern with programming and software development at large; this is an important issue because it directly challenges the production mode of the hacker ethic, which is often present in the open, collaborative practices of the software culture. The emergence of software engineering attempted to respond to the soft-

ware crisis directly and, in doing so, pursued the professionalization of the field. As Nathan Ensmenger pointed out in his study on the creation of the computer expert, the entangled elements of the software crisis were already present in the mid-1960s, namely, the criticism of artisanal practices in programming and a gap between corporate and academic expectations in the field. The last element involves confrontations between experts and management and, consequently, software projects not meeting time and costs constraints [Ensmenger, 2010, 24]. These tensions led to the pivotal moment for software engineering, a conference in 1968 on the subject held in Garmisch, Germany, and sponsored by the North Atlantic Treaty Organization (NATO). The second section focuses on the paradigm shift of software production and distribution, assessing the agency of source code (and other artifacts) in software development and contrasting it against the emerging prohibitions in the community where code sharing is customary. Because software was becoming an asset after a hardware unbundling process, practices common to other industries, such as commercial confidentiality, were applied to source code, discouraging its sharing practices. That is not to say that software had no value before; however, several open collaboration practices in software development were affected by the emerging economy based solely on software rather than hardware. Finally, the third section presents the departure taken by the free software concept as an answer to the set of tensions already described. Although collaborative practices and source code sharing in software development are not new, the existence of a set of principles that systematize them marks a milestone in software culture and history whose consequences are still unknown. Free software also represents a change in the perception of the hacker culture, inaugurating a new period in which professionals are aware of the importance of their practices and their role in the economic and social effects of software. It can be said that the free software, philosophically and technically, embodies an entire ethos that is often (and mistakenly) regarded as opposed to modern commercial software. It constitutes a large part of the modern metaphor of the free/open, which has been so enthusiastically adopted by different disciplines.

The period covered by the three proposed sections spans approximately fifteen years, beginning in 1968 with the Garmisch conference on software engineering and ending in 1983 with the free software manifesto by Richard Stallman. During this period, several technical advances took place that shaped the field and improved open, collaborative practices. This was the period when networks took off with Arpanet and the personal computer entered the industry almost out of nowhere. New programming paradigms, such as structured or object-oriented programming, also came into the scene. This

period also clearly outlines the stretch between the modes of software production and the effects of software commoditization that produced a shift in the development and distribution politics of the field. This last statement again stresses the change of direction expressed by the metaphor of the software crisis.

3.4.1 Software engineering: a problematic discipline

The idea of a software crisis was the trigger to start a reflective process on software building techniques leading to the establishment of software engineering as a discipline. As already suggested, software engineering tried to formalize a growing field, which was becoming a proper industry, however, full of intuitive procedures, particularly concerning the central act (for software development) of programming. This scenario contextualized the seminal 1968 conference held in Garmisch, Germany, which was heralded as the inauguration of Software Engineering. The software crisis permeates almost all of the statements in the famous report on this conference, indicating the need for a solution given the increasing presence of software in our daily lives (if noticeable in 1968, let alone today) [Naur and (Eds.), 1969, 9]. The diversity of approaches proposed in the conference is also evident in the conversational style used in various parts of the report. The participants, including some that would become key players in the establishment of software engineering as a field, provided their insights on the problems that plagued software development, which were divided into sections that somewhat resembled the cascading sequence of a development model. As I will explain in the next paragraph, this multiplicity of approaches represented an unresolved tension that surfaced both as a new re-edition of long-standing discussions on the nature of technology in a subject field as fluid as software and a statement of the inability of software engineering to provide a single coherent solution to the increasing issues of software production. As observed by Mahoney in his account of software engineering history, "[t]o what extent computer science could replace the rule of thumb in software production was precisely the question at the NATO conferences (and, as noted, it remains a question)" [Mahoney, 2004, 14]. This observation reminds us of the persistence of this question posed in 1968, which remains unanswered (as is the case of the alluded software crisis), and that the desired systematization process of software production was not accomplished.

To briefly illustrate the struggle between formalization and hands-on pragmatics while providing a connection to the previous sections, it is fitting to consider the time-sharing systems saga before the Garmisch conference.

First, because it is the direct antecedent of the meeting, at the same time, a notable example (for example, Multics and OS/360) of the dilapidation of time and money, "the transition from batch processing to time-sharing systems was vastly more difficult that anyone had anticipated" [Wirth, 2008, 33]. Second, because the successful outcomes of the time-sharing effort, particularly the ITS or Unix operating systems, are due more to a hacker style approach than to a coordinated software effort. Take, for instance, the relevance and close connection of Unix and the C programming language. The latter provides freedom and flexibility to the programmer because of its ability to access low-level routines, but precisely by doing so, it disfavors a multilayer style language with adequate verification and defined interfaces, which actually represent a leap backward in the structured programming and high-level languages advocated by software engineering [Wirth, 2008, 34,35]. Summing up, the ongoing crisis attests that the main concern of software engineering still lives on. The decoupled realities of the academia (where software engineering was proposed) and industry (where pragmatism still reigns) are a demonstration of this initial tension, poignantly epitomized in Dijkstra's (a key actor of software engineering) statement, "software engineering is programming for those who can't" [Wirth, 2008, 36].

Software engineering was and is an amalgam of different recommendations on how to face the software crisis and produce quality software using good developing practices. Because it was a new discipline, without any past or precise referents, when it was proposed, the understandable action was to employ the metaphors from other well-established fields. Herein lies one of the main elements to understand the nature of software engineering, its shortcomings, and also its achievements. In this order of ideas and to frame software engineering in this chapter and, in general, the purpose of this work, we need to briefly examine the disciplinary tensions within software engineering, its relationship with the course of hacker ethics in open collaboration practices, and some of the pragmatic outcomes of the tense discursive field of software engineering.

Software engineering is defined in scientific, mathematical, and engineering principles [Mahoney, 2004, 8]. More precisely, it is based on conceptions such as applied science or other engineering fields, such as mechanical or industrial engineering [Mahoney, 2004, 9]. As the report from the Garmisch conference states, "[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 (Eds.), 1969,

13]. This statement of principle merely reflects the hybrid nature of software engineering as an elusive discipline between theory and practice, where its practitioners never agreed on what it really was [Mahoney, 2004, 8]. It also reworks some of the tropes already covered in Chapter 2, such as the difficulty between the conception of technology as applied science or the method of engineering and how mathematics is associated with it. The nature of software as technology and its symbolical substrate add complexity to the agenda of software engineering. Take, for instance, mathematics, which is a fundamental part of the idea of applied science.

There is a strong precedent of software engineering using mathematical reasoning as a tool to obtain formality. However, this reasoning comes into question when we consider that programming and computing are different tasks [Wirth, 2008, 33]. Even as Daylight extensively argues, viewing the computer as the logical result of something that started with logic and mathematics is deceptive (a recurrent notion that slights engineers) [Daylight, 2012, 1-3]. Turning back to the hacker-related vein of the discussion, the hacker approach to software development questions this mathematical view of software; this can be attested by the hacker's evaluation of the term "computer science" and what it embodies. This is only one instance of the gap between the *supposed* mathematical origins of computer languages (and their associated model of the computer to *compute*) and the data storage and communication features required by actual applications [Wirth, 2008, 33]. As an example, take this account from Stoyan quoted by Daylight in his argument against the excessive formalization in software engineering, "nobody, including McCarthy himself, seriously bases his programming on the concept of mathematical function" [Daylight, 2012, 2]. In fact, McCarthy, the alleged father of LISP, one of the functional languages considered a direct distillate of lambda calculus, required the help of several hackers to make the language reality in different computer architectures [Levy, 1994, 47,79]. The issues of treating programming as a subject of mathematical reasoning are challenging, considering that "the craft of programming turns into 'hacking' in many cases" [Wirth, 2008, 35]. However, it would be misleading to think in a dichotomy between the need for formalization maintained since the Garmisch conference and the fact that, for the attendees, "[p]rogramming is still too much of an artistic endeavour" [Naur and (Eds.), 1969, 15].

Although for expository reasons, the tension between the *hacker* and the *planner* has been presented as an opposition, current practices in software development have shown that the search for formalization, advocated by software engineering, has indeed produced several contributions that now

are part of the modern concept of software, no matter who is developing it and how. Specifically, techniques and tools like structured programming and support software, which are currently widely used, are now taken for granted (although brand new in the times of the Garmisch conference). However, they are based on controversial assumptions on the nature of software. Structured programming, indebted to the pioneer Edsger W. Dijkstra, is regarded as a direct reaction to the software crisis. It was presented at the Garmisch follow-up conference on software engineering a year later, in 1969 [Daylight, 2012, 117]. Admittedly, even Dijkstra was well aware of the shortcomings of translating a correct but small mathematical demonstration into full-size programs. He recognized the inability to infer the proofs in a larger scope, which would involve programs with a number of lines of code exceeding the size of proven code [Dijkstra, 1972, 1]. Based on the opinion that "the art of programming is the art of organizing complexity" [Dijkstra, 1972, 6], Dijkstra followed a sound mathematical reasoning, including enumeration, mathematical induction, and abstraction as argumentative devices, to present structured programming as a way to address the growing complexity of software that concerned early software (and current) engineering professionals. Even though early computers like the EDSAC featured concepts like the closed subroutine, this responded to a convenience provided to the programmer lacking in-depth analysis, which left the whole program acting as a "single homogeneous store" [Dijkstra, 1972, 46].

Here, a small break must be made to retake the discussion on the linearity of source code considered as text. The introduction of structured programming went a step beyond the break in linearity represented by the input of interactive computing compared to batch processing. One of the main issues discussed in the early software crisis narrative concerned the problem unleashed by companies rushing into the concept of time-sharing (a development involving interactivity) without preparation. As pointed out by Wirth in an interview with Daylight, IBM had "thousands of programmers working on OS/360 alone" [Daylight, 2012, 117]. Structured programming suited the upcoming shift in programming paradigms to event responses well, replacing a lineal finite-state machine model with discrete events [Wirth, 2008, 37]. Besides the changes in programming practices, another important source of change was the software tools, as well as the organization of the software production as standard manufacture.

Clearly, if new programming techniques, partially shaped by mathematical reasoning result baffling, the metaphor of the production line applied to software is not less daunting. As suggested previously, because software was

becoming an economic force, models applied successfully to other industries emerged as an answer to the software crisis, completing the web of multiple approaches that is software engineering. Industrial and mechanical engineering were in particular important forces that shaped, from the organizational and industrialist perspective, the mass production software metaphor, with mechanical engineering providing the manufacture and interchangeable parts paradigm [Mahoney, 2004, 11], and industrial engineering contributing with management and the continuation of "scientific management", and Taylorism [Mahoney, 2004, 12,13]. Understandably, this framework is not unchallenged, as suggested in this quote by McIlroy in the Garmisch report "[o]f course mass production, in the sense of limitless replication of prototype, is trivial for software" [Naur and (Eds.), 1969, 139]. There is a tempting trend to conceive the possibility of software development as an assembly line; however, it would be more accurate to recognize that "effective management of software production lies somewhere between craft production and mass production" [Mahoney, 2004, 16]. Despite this problematic panorama, the narrative of mass-produced software was not at all fruitless, coupled with structured programming; it explains several developments in this well-known difficulty among the corpus of the software culture.

Software engineering is of crucial importance because it brings into play the general concern of technology as an originary technicity. By deduction, it allows a similar light to be cast on software (augmented by its symbolic tropes), thus, explaining the elusive nature of software as a technology, which is one of the fundamental premises of software studies. Although the forces surrounding software production are sometimes presented as oppositions, it should be clear that software's complexity transcends dichotomies; it is, instead, a network of artifacts, histories, and communities interacting within. From an open collaboration in software production perspective, software engineering's role is fundamental, given the crisis narrative and the history of its emergence in which sharing practices were questioned, as well as the development of techniques and artifacts that established a common ground for thinking critically about the logic of collaboration, alluded to in current software theory (technically and sociologically speaking). That said, it often happens that pragmatism is the force that apparently reconciles divergent readings, usually, in the form of tools. For example, the original concern regarding better tools and methodologies discussed at Garmisch, in part, compatible with the idea of tools applied to production flow. However, during the same conference, it was stated 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 (Eds.), 1969, 10]. Despite this problematic nature, the idea of software tools brought about developments in the form of better compilers, CASE tools, and techniques such as structured programming, among others. The combination of these tools and techniques produced the extensive use of modules in the creation of software and, as Wirth directly expressed "modularization, probably constituted the most important contribution to software engineering, that is, the construction of systems by large groups of people" [Wirth, 2008, 36]. From here, collaboration (open or not) follows as the inevitable consequence of the course of software engineering.

To conclude, consider again the Unix operating system and its tool implementation principle, described in the mentioned "Unix philosophy". Historically speaking, Unix has a prominent status because of its connection to the appearance of the time-sharing paradigm, its embracing of the hacker ethics, its close relationship with the free/open software metaphor, and its extensive use in computer networking. Moreover, "Unix is leading a model of the notion of software tools and of a programmer's bench" [Mahoney, 2004, 16]. This last observation is key to understanding the discursive connections between software engineering, open collaboration, and tools. If the metaphor of the line of production is contested when assessing software, the Unix programmer's workbench fully embraces the idea of the workshop rather than the assembly line; this, in turn, hints again at the craft-skilled workforce as the producers of software [Mahoney, 2004, 16]. This metaphor conveniently approximates current speculations of a hybrid manufacturing software culture model workshop, as in the maker culture.

3.4.2 The time of the shifting paradigms

Another concern in the 1968 Garmisch report on software engineering, which was relevant enough to produce a special section in the report, was the discussion on the cost of software. When it became clear that software had a financial value in itself and that the transition from hardware to applications was conceivable, the resulting discussion revolved around arguments against or in support of software pricing. The arguments in support involved reasonings such as benefiting the customer with freedom of choice, improving the quality of software due to competition, allowing new players producing good software to participate in the game (such as the academia), among others [Naur and (Eds.), 1969, 76]. However, and although the overall result of the discussion favored pricing, there was an argument against the case that sounds very familiar and that, in its phrasing, predates the common rhetoric of movements such as free software by fifteen years, "[s]oftware belongs to the

world of ideas, like music and mathematics, and should be treated accordingly" [Naur and (Eds.), 1969, 77]. This statement cues a brief introduction of this section, which deals not only with the price of software but also with the secrecy of source code.

As it has been presented, open, collaborative practices in the software culture depended largely on the agency of source code, amplified by the free access to it. However, as a result of the software crisis and the emergence of software engineering, if production metaphors from other fields were being applied –not seamlessly– to software development, then source code would also be affected by these forces. In short, source coded was now considered one of software's key assets and, therefore, should be protected by confidentiality. The outcome was familiar in the history of software, questioning practices such as the ones described in the SHARE community, the MAC project laboratory, or the early Unix development process, which were, until that moment, taken for granted.

Therefore, I selected three specific examples that show this paradigm shift that would become preponderant in the software realm. The examples are the antitrust case against IBM, which resulted in the process of unbundling that separated the costs of hardware from software, the change in the license terms of the Unix operating system by AT&T, and the now-famous bout of a very young Bill Gates against software "theft". These events occurred in the timeframe following the software crisis motive and raised major counterreactions to the software culture.

There is the common understanding that the IBM's decision in 1969 to unbundle its hardware from software and services was a turning point in the history of the software industry. At the time, the company stated that the decision "gives rise to a multibillion-dollar software and services industry" [IBM, 1969]. Supporting this view, Berry said that IBM's unbundling of software from hardware represents a "crucial inflection point" that helped to "cement the foundations of a software product market" [Berry, 2008, 61]. This year was also heralded as the end of the decade when software engineering took off, the academia opened computer sciences departments and several developments were made that "liberated the computer from its primary role as a calculating engine and headed it toward a future centered on information management and networked communications",[29] [Lowood, 2001, 145]. This

[29]The quote specifically refers to the work of Douglas Engelbart, but it is used in the original as an epitome of the era.

perception is reinforced in the same text by quoting software historian Paul Edwards, indicating the parallel shift in computer historiography from "machine calculation" to a new set of writings based on "machine logic" (meaning software) [Lowood, 2001, 146]. As is often the case, some accounts diverted from this often featured narrative of important events, focusing on the existence of some well-established software companies predating 1969 [Johnson, 1998, 36]. However it should be acknowledged that there was hardly a software industry prior to 1967 because software was freely distributed and "bundled" with hardware or written by independent contractors to meet a very specific demand [Campbell-Kelly, 2004, 29].

Nonetheless, IBMs decision to charge for software separately from hardware must be remembered as a significant event given that IBM was the major player in the computer business at the time and, as explained in this chapter, because the company had promoted several open-source practices in the past, a change in its procedures would significantly affect the entire industry. Beyond a discussion on the existence of a software industry *per se*, at the time, it could be argued that the entire computer sector had already seen the potential of software as a source of profit, which explains the investigation of monopolistic practices opened in 1968 against IBM by the Department of Justice of the United States (instigated by other companies). This event activated an internal process at IBM that precipitated the unbundling decision, although the company stated that the two events were unrelated. Whether this juridical process influenced IBM's decision to unbundle still remains an open question. It could be conjectured that while providing integrated solutions had granted IBM its dominant position, to the dissatisfaction of other early competitors in the software field, the increasing costs of producing software and the emergence of their S/360 platform as a *de facto* standard, which facilitated the sale of software for the platform with no modifications, had probably led the company in a similar direction. Lastly, given IBM's size and influence, unbundling represented a paradigm shift in collaborative practices in software production. Until then, IBM had developed software in-house to be offered free of charge to support its business model or software developed in the field with clients who well aware of the benefits of sharing results [Grad, 2002, 66]. These contributions from support engineers on the field were incorporated into a software library maintained by IBM and were made available to other customers with the help of related user groups such as SHARE and GUIDE [Johnson, 1998, 37]. Following the unbundling decision, this model was reexamined and a work group recommended charging for the most promising applications and setting a transition period to leave behind the public domain code and honor existing contracts. This led to

the establishment of an entirely new software business model that ended an era and outlined a completely different future for the company, "IBM would copyright each program. In addition, an individual usage license with each customer would preclude the customer from copying or selling the program or its usage rights to anyone else. This license included trade secret provisions" [Grad, 2002, 68].

The distribution model (and its development model) of the Unix operating system was also affected by a paradigm shift because of a decision made in 1979 by the then owner of the operating system, AT&T. Like IBM, the event had several consequences for open collaborative practices, considering Unix's influence, although sometimes unnoticed by the normal user, in the existing operating system landscape. And also like IBM, the reason for the change in operating system distribution had a legal connotation that in itself reflects an era in which the software business took off. Chronologically speaking, to frame the history of Unix, it is necessary to go back to 1949, when AT&T and its subsidiary Western Electric underwent an antitrust proceeding that ended in 1956. The final ruling of the process advised –so to speak– that the companies should to stick to their telephone business and prevented them from pursuing economic interests outside of this sphere [Salus, 1994, 56,57]. Although in 1956 there was no reference to software –the word had not been invented– AT&T took a conservative approach to comply with the demand for, specifically speaking, the kind of software represented by operating systems. It entered into a joint effort with Multics, which, by effects of the decision, was presented as research. Later, when Unix was requested because no profit could be obtained, the licenses were granted at a nominal fee[30].In part, this shows Unix's hybrid nature; even though it was the product of a private company, it nurtured a culture of sharing and a development model that would not have been possible if it had been commercially marketed. In fact, and as explained previously in this chapter, the Unix community features hacker culture principles more visible in the academia (as in MIT's MAC project) than in companies, fostering a culture where the operating system would thrive thanks to collaboration. During Unix's first decade (1969 to 1979), its informal development model could be classified as open-source. It relied on the agency of shared code and an international community, "something was created at BTL (Bell Telephone Laboratory).

[30]Detailing the legal specifics issued by the Department of Justice exceeds the scope here, but, according to Unix historian Peter Salus, AT&T,Western Electric and Bell Laboratories were under a special regime that "resulted in a much more rapid dissemination of technology than would otherwise have been possible under our patent law" [Salus, 1994, 58].

It was distributed in source form. A user in the UK created something from it. Another user in California improved on both the original and the UK version. It was distributed to the community at cost. The improved version was incorporated into the next BTL release" [Salus, 1994, 143]. This statement shows how the community filled the void left by AT&T and its lack of interest in Unix, which was not a viable source of revenue (as the joke went "no support, pay in advance"), as well as what was about to change in 1979.

This year, AT&T released the Unix V7 (seventh version) and decided to change the terms of the Unix license. From then on, not only was the fee for various licenses increased but the access to the source code was compromised, "[w]hen AT&T released Version 7, it began to realize that UNIX was a valuable commercial product, so it issued Version 7 with a license that prohibited the source code from being studied in courses, in order to avoid its status as a trade secret"[31] [Salus, 2008, 38]. This was only one step on the road that led finally, in 1984, to the divestment of the AT&T companies, which freed them from monopoly concerns and enabled the full entrance of Unix into the commercial market. However, and despite the new restrictions of the operating system, the proliferation of Unix and its many bifurcations with access to its source code were evident (attested by consulting the somewhat chaotic Unix version tree) and indicated a diverse and burgeoning landscape outside the control of AT&T. Particularly noteworthy is the development branch started in Berkeley, which would later be known as BSD (Berkeley Software Distribution). Berkeley would contribute several developments that would be incorporated into AT&T's codebase. BSD attained DARPA's backing thanks to the reputation it obtained through its remarkable role in disseminating networks (particularly ARPANET) with its early implementation of the TCP/IP protocol. Because BSD used AT&T code, it required a license; however, when the prices rose, Berkeley began a cleaning process that ended in the nineties with the release of the fully distributable (binary and source code) BSD Unix operating system [McKusick, 1999, 25,26].

To conclude the case of Unix, the early history of this operating system and the changes that it underwent are not only an excellent example of collaboration in software development but also of the tensions between academia, industry, and the hacker culture on the verge of the commodification of software. Unix's licensing changes also generated a series of reactions whose

[31] Even though this quote indicates that it was " (...) being studied in courses", what is important is the new status of source code as a trade secret, which poses a problem for collaborative practices and system improvement based on the free flow of source code.

waves still reach us today; for instance, Andy Tanenbaum's academic Unix-like Minix or the idea promoted by Richard Stallman to implement a completely free system similar to Unix at the emergence of the free software movement. Also, considering the case of Linux and the inheritors of the BSD tradition (such as FreeBSD or NetBSD), it could be said that Unix set an important model for software, not only as a product but also as a process. And although a series of lawsuits plagued the Unix market in the 1980s and 1990s (concerning the ownership of source code), in the end, an entire collaborative production took place despite the affectation of Unix source code as a trade secret, "[t]he history of the Unix system and the BSD system, in particular, had shown the power of making the source available to the users. Instead of passively using the system, they actively worked to fix bugs, improve performance and functionality, and even add completely new features" [McKusick, 1999, 25].

As the final example of this paradigm shift in the 1970s, we have the infamous "Open Letter to the Hobbyists"[32] [Gates, 1976] written by Bill Gates at a time when the market for microcomputers and their software was still incipient. This letter is of paramount importance because it is at the heart of the turmoil between two models of production that characterized the commodification of software and already displayed some of the shared arguments of the critics of collaborative practices. Its historical significance is also remarkable considering the prominent status that Microsoft would gain in the coming PC business and, as Berry expressed, the direct attack on "the hacker conventions and norms of freely sharing software and ideas" [Berry, 2008, 100]. To start, the letter acknowledges the existence and potential of a so-called "hobbyist market" and states the intention of the still embryonic company to step in into it. Two statements must be presented to frame this claim; first, that the hobbyist must be equated with the hacker, an easy comparison considering the role of the Californian hobbyist community, which Levy labeled as "hardware hackers" [Levy, 1994, 149], in the introduction of the personal computer. In the second place, the reissue of the symbiotic relationship between hardware and software in a new field –the personal computer was still considered a hobbyist issue– by pointing out that "[w]ithout good software and an owner who understands programming, a hobby computer is wasted" [Gates, 1976]. Gates then lists the costs (mainly employee and computer time) that he and his business partner (Paul Allen) incurred

[32]Due to space constraints, I will not quote the entire letter but rather proceed to discuss it in a linear fashion, providing an exact quote when necessary. The letter is available from several internet sources.

in developing the Altair BASIC laying the foundation for a classic argument, "[t]he feedback we have gotten from the hundreds of people who say they are using BASIC has all been positive. Two surprising things are apparent, however, 1) Most of these users never bought BASIC (less than 10% of all Altair owners have bought BASIC), and 2) The amount of royalties we have received from sales to hobbyists makes the time spent on Altair BASIC worth less than $2 an hour. Why is this? As the majority of hobbyists must be aware, most of you steal your software. Hardware must be paid for, but software is something to share. Who cares if the people who worked on it get paid?" [Gates, 1976]. In retrospect, this statement signals a self-evident fact, which is that developing software is costly; however, he then proceeds to accuse a whole community of *theft*, offering a rather narrow vision of an until then common practice (sharing software).

Furthermore, it could be argued that sharing code was a precondition to extensive feedback that led to the improvement of software (in benefit of the company). However, as the letter would show, this part of computing and software history was simply ignored: "[o]ne thing you don't do by stealing software is get back at MITS for some problem you may have had... [o]ne thing you do is prevent good software from being written. Who can afford to do professional work for nothing?" [Gates, 1976]. This statement expresses a nonexistent incompatibility between the quality software and the sharing of software (and collaborative practices based on the free flow of source code in general); it was just one instance of a common movement in techno-economic determinism, where trade secrets (affecting software in source code form) or copyright law (affecting software in binary form) are applied as a mechanism to ensure the need required for the capitalist mode of production [Chopra and Dexter, 2008, 18,19]. Again, this shows a dichotomy between two ways of developing software. It could be read from a philosophical (human motivation) or a more mundane or pragmatic perspective; that is, disagreement on how the computer industry should develop [Weber, 2004, 37]. Suffice it to say that not only history (SHARE, ITS, or Unix) but the events to come would expose the weakness of Gates' argument.

Moreover, how the software development industry developed following both paths (proprietary software and collaborative-based software) presents software as a negotiation field. Faced with Gates' obvious attack, in fact, his letter was first addressed to and published in the newsletter of the now famous "Homebrew Computer Club", a west coast community that fostered the appearance of the Personal Computer or companies such as Apple, the hobbyists, or hardware hacker community, met the challenge by releasing a

version of BASIC developed by the community. A particular reaction is presented to wrap up the analysis of Gates' letter. The correspondence section of the July 1976 edition of SIGPLAN featured a statement by Jim C. Warren Jr.[33] in which he mentions Gates openly, offering a solution to the conflict of theft posed by him. He stated, "[w]hen software is free, or so inexpensive that it's easier to pay for it than to duplicate it, then it won't be 'stolen'" [Warren-Jr., 1976, 1]. Several examples followed, notably, the Tiny BASIC, which was available for various platforms at the cost of $5, and even if that was too high, the annotated source code and implementation instructions were published, inviting to the community to "retype it and reassemble it. No one will yell, 'thief'" [Warren-Jr., 1976, 1]. He also announced a new Journal as a medium for sharing among the hobbyist community (including an index of other hobbyist publications) in which, a few months before, a description of the TINY BASIC was published, labeling it as a "participatory design" [Dennis Allison, 1976].

These three examples (IBM, Unix and hobbyist) highlight several concerns related to the change in the software industry in the 1970s and how collaborative practices were affected by it. Each one corresponds to a category in hardware classification, namely mainframes (IBM), minicomputers (Unix), and microcomputers or personal computers (Bill Gate's letter to the hobbyist), revealing the entangled nature of hardware and software. Despite collaborative practices being questioned in every case, communities would continue to share source code, and build and improve software based on the potential provided by this approach. Perhaps, the reactions to the changes were more notable in the case of Unix or personal software, considering that these sectors were still in the making –especially the personal computer– compared to the rigor of IBM's monopoly (although the SHARE user group continued its tradition). The example of the personal computer and its software would be of undeniable importance, considering this industry's enormous impact, which catapulted the computer to a communication device, a standard device for the layperson (perhaps granting Bill Gates and his letter a legendary folklore status). Its relevance increased when the hobbyist community, connected to the personal computer, joined forces with the more academic influenced Unix hacker community, giving birth to the self-aware hacker community embodied by the free software movement.

[33]Warren was the editor of the "Dobb's Journal of Computer Calisthenics & Orthodontiathe", a publication of the PCC (People's Computer Company), an organization also important in the creation of personal computer and the early adoption of computer networks.

3.4.3 Second iteration: Free Software and self-awareness

Although the concept of free software and the movement associated with it originally appeared in the United States in the '80s, the ideas it grouped were based on a set of collaborative practices present since the dawn of computing and software. Therefore, the prevailing narrative of assessing free software as a single reaction against proprietary software loses the interwoven threads behind the creation of free software. In short, although the increased commodification of software combined with the access to inexpensive personal hardware played a role, the forces behind free software are more complex and represent some of the issues discussed throughout this work. The obvious prominence of free software lies not only in its historical background but also in the consequences it generates, which have made free software an important template for open collaboration practices and a cornerstone of modern software history. The historical context is then considered from the very emergence of free software.

The original release of the concept of free software can be traced to 1983, highlighted particularly by Richard M. Stallman, at the time, and employee of the MIT AI Lab, and a self-declared "hacker". This observation leads to some interesting points. First, it connects free software to a former tradition like the hacker culture (particularly at MIT). Second, it places an individual at the forefront of an influential software event, somehow bringing up the problematic question again of labor and programming in software, seen as art, from one side, and an object systematization and division from another. From this personal perspective, some events –now belonging to the hacker folklore– in the work of Richard Stallman reveal not only the personal nature of the emergence of free software but also the struggles brought on hacker practices by the shift in software development and distribution patterns. For instance, the story of a printer given to the MIT AI Lab by Xerox, whose code was closed and no longer accessible, preventing Stallman and others from introducing the desired function (which was, reporting printer's status directly to the user's workstation) into the device. This change in the manufacturer's policy posed a dilemma to the hacker ethos and reduced one of the key principles of collaborative software development *"[e]in praktisches Problem stellt sich. Die Lösung besteht darin, eine bestehende Software um eine Funktion zu erweitern. Früher hätte man den Autor der Software um den Quellcode gebeten und hätte diesen fortgeschrieben –die Technologie wäre für alle Betroffenen nützlicher geworden. Doch jetzt stand vor dem Quellecode und vor einer Kooperation von Programmieren eine Mauer namens »geistiges Eigentum«"* [Grassmuck, 2004, 222]. This passage discusses

the concept of software ownership, enforced through licenses (final users) and non-disclosure agreements (developers), which was clearly a new phase in the computing world, which, as noted by Grassmuck, was an obscure one [Grassmuck, 2004, 221].

Another example, related to this change in the legal nature of source code and its effect on incremental development, was the case of the Lisp machines. Lisp was the language of choice of MIT's breed of hackers and the development of compatible AI (Artificial Intelligence). In the early 1980s, two new startups emerged from the MIT AI lab, Symbolics, and LMI (Lisp Machines Inc.). These companies were committed to the manufacture of Lisp machines, a computer with specialized hardware to run the Lisp language and its applications. It is said that, at the time, the hacker culture of the MIT AI lab (heir of the MAC project) was falling apart. Hackers were disbanding to join either of the two new companies, developing a delicate balance in which MIT retained the code base of the Lisp machines while both rivaling groups contributed their improvements to the same [Chopra and Dexter, 2008, 12]. Because Symbolics was clearly more commercial, with more funds and supporting the proprietary model, Stallman decided to work with LMI (allegedly a *hacker company*) to ensure that both of them remained competitive. Legend has it that in just over a year, Stallman alone outcoded Symbolics programmers. He replicated every improvement introduced by them, making it available to LMI through the codebase at MIT, to which he had access as an AI Lab employee. This enterprise certainly sealed Stallman's legendary status [Levy, 1994, X]. The unsustainable effort aside, it was Stallman's character that excelled, showing him as a man of principles, with a strong will to preserve his beloved and fading hacker culture[34]. When new computers with proprietary operating systems arrived at the MIT AI Lab, Stallman knew that actions were due and that total independence was a must in what he was about to begin.

Stallman abandoned the MIT AI Lab to embark on a kind of techno-philosophical crusade whose outcome had a huge impact. Free from any possible intellectual property disputes with his former employer, Stallman decided that in order to preserve his ethical position and promote the technological benefits of sharing software and build on it, the proper course of action was first to attempt a "free" operating system. Thus, this movement could be framed as hybrid "pragmatic idealism", a term used by Stallman himself in an essay

[34]This story is mentioned because is part of the free software narrative. Nevertheless reinforces the problematic conception of software history as a result of individual efforts

he wrote, and a label that captured the attention of some critics. Weber, for example, highlights the combination of moral and pragmatism in the origin of free software [Weber, 2004, 47]. Berry directly addresses this dual nature "for Stallman it was anathema to programming ethics to hide and refuse to share source code, particularly when it would help another person fix a problem" [Berry, 2008, 110]. In this statement, he specifically points to source code as the central artifact in the discussion. Furthermore, and as previously hinted, Stallman's technical and idealistic stance is largely based on the hacker culture, which represents an approach that could be described as "clearly pragmatic from a technical perspective, [that] embodied a broader philosophical and social goal: to replicate and disseminate the ideals of freedom and cooperation that characterized much of hacker culture, particularly that of the golden days of the MIT AI Lab" [Chopra and Dexter, 2008, 13].

Perhaps this double condition (pragmatism and idealism)is best portrayed by the Stallman's own production. On the one hand, he was the author of celebrated programs such as the Emacs editor or the GNU C compiler (dubbed "GCC") and, therefore, considered a top coder. On the other, he is also known for the direct prose of his writings, in which he fiercely advocates for the relevance of (free) software and its cultural significance given the current state of things. Three of his texts are briefly discussed in order to evaluate his main ideas and their incremental evolution.

The first document was later called the "original announcement" (1983); it inaugurates a new era in which a network platform like USENET was used to share Stallman's intention of building a free Unix, called "GNU"[35] with the community. The announcement openly addressed the collaborative aspect of the enterprise, asking "computer manufacturers for donations of machines and money" [Stallman, 1983] but mainly asking "individuals for donations of programs and work", which demonstrated the type of development proposed. It went on to explain that "[i]ndividual programmers can contribute by writing a compatible duplicate of some Unix utility and giving it to me. For most projects, such part-time distributed work would be difficult to coordinate; the independently-written parts would not work together. However, for the particular task of replacing Unix, this problem is absent. Most interface specifications are fixed by Unix compatibility. If each contribution works with the rest of Unix, it will probably work with

[35]The acronym GNU, which, stands for "GNU is not Unix", is usually described as a typical hacker-style recursive acronym (and a pun on Unix). It also points to the idea of extending Unix freely, given the already described changes in the operating system policies and the to take distance from what Unix was becoming.

the rest of GNU" [Stallman, 1983]. This particular phrasing is of paramount importance because in recognizing the suitability of Unix for collaboration based on its modularity, it traces a method of distributed development that is based –presumably– on networks.

The second document is actually a manifesto, hence the name "GNU manifesto". This document further elaborated on the arguments outlined in the USENET announcement, with clearer objectives thanks to the fact that the project was already underway. For example, the Unix argument was worded almost exactly as in the initial announcement. However, the ethical aspect of the project was reinforced by the following statement "I consider that the Golden Rule requires that if I like a program, I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement"[36] [Stallman, 1985]. Two additional aspects point directly to the specificity and materiality brought to the project by the experience of the previous two years and the tools developed, so far as informed by the manifesto. If the hardware platform was not clear in the initial announcement, the 1985 manifesto clearly states a preference for the 68000/16000 class processor (with virtual memory), in a move that exemplified the surge in workstation architecture at that time, as well as a simple, pragmatic choice because "they are the easiest machines to make it run on" [Stallman, 1985]. Although both the manifesto and the first announcement announce the intention of implementing network protocols and establish connecting capabilities as a must-have feature, the manifesto specifically mentions Internet protocols, foreseeing the coming change in computation [Stallman, 1985].

The last of the three seminal and fundamental texts in the history of free software is the GNU General Public License, perhaps the most influential of them all. This document not only distills in legal form the desire of protecting the programs considered free software from additional appropriations by third-party partners, but it mainly exalts source code as one of the key artifacts in the creation of a collaborative agency through guaranteed access and modification rights, let alone the crucial move of making its derivative works also free software [Free Software Foundation, 1989]. The GPL license is one of the clearest examples of the so-called "copyleft", that although not

[36]In both communications, Stallman openly dismisses the software licenses and non disclosure agreements, signs of the increasing commodification of software.

entirely new (see the previous section), marked a final drive in the peer-produced software model [Berry, 2008, 115]. Moreover, the GPL is regarded as a legal hack, a clever way of protecting a set of values using the same tools, namely copyright, as the proprietary software model. The fact that newer versions of this license are still widely used by several programs (not only GNU) confirms its relevance.

To conclude the evaluation of free software, I connect a few points to round out its understanding in a broader context, taking into account the already mentioned pragmatic idealism that characterized Stallman's work and the foundation he created, the Free Software Foundation or FSF (1985), to organize the corresponding activities. Weber's observation of Stallman stands out. He stated that for Stallman, software was not just a tool; instead, it was an expression of creativity and an artifact of community [Weber, 2004, 47]. Following this train of thought, and drawing from the discussion in the second chapter, I will argue that the free software construct is, in fact, an instantiation of originary technicity in the field of software as a technology. Although, as stated previously, free software is not entirely defined as an opposition to proprietary software, a common simplification that oversees the presence of collaboration throughout the history of software. The duality of access to the software source code substrate, that is, whether it is closed or open-source, directly implies support for the former claim. Closed source significantly limits the collaborative agency of source code, reducing software's possibilities, and leaving it open to instrumentalization. On the other hand, by granting full access to source code and increasing its collaborative agency through a legal framework, free software broadens its panorama of possible uses and interpretations of the software. In this way, some of its uses can be merely instrumental. However, other uses enhance the collaborative ethos of a socio-technical system based on peer-to-peer collaboration exercised by a community. A closer relationship with the technology can be obtained, one in which, to use Heidegger's words, the "essence" of technology (which can be interpreted as the source code in the case of software) is available to its users to play with it at different levels. This vision is compatible with the supposed position of programming as an art, possibly a more satisfactory evaluation of software that questions the idea of software as a tool and offers a field in which software is a medium to individual and collective expression. Stallman states in his writings that "copying all or parts of a program is as natural to a program as breathing"[Stallman, 2002, 36] or that "programming has an irresistible fascination for some people, usually the people who are best at it"[Stallman, 2002, 40], pointing this out as a very human part of software.

Nonetheless, the field of free software is still a tense scenario, evidenced by the unstable label of "pragmatic idealism". For example, the decision to select Unix instead of developing a new and better system obeyed to the existence of a community and the modularity of Unix (hence, its pragmatism), which although it was recognized as a system, it was far from being ideal [Stallman, 1985]. Similarly, the copying and remixing ethos of free software, visible at multiple levels[37], resulted from the point of view of programming by favoring the use of already tested designs and avoid research projects. In other words, not trying to reinvent the wheel [Moody, 2002, 27]. This decision was crucial and perhaps partly explains the insistence in reusing other kernel designs for the GNU system instead of producing a new one, a move that resulted in the inability of the GNU project to be completed as initially intended. In the same vein, Berry observes that modern software development shows the struggling relationship between the skilled individual (the programmer) and the company's interest, a tension that became public and was placed under political scrutiny by the free software movement [Berry, 2008, 110]. Finally, it can be said that equally valuable to the programs and legal aids developed by the free software movement, in trying to preserve the hacker culture and its mode of software production, Stallman brought the community a needed self-awareness and the first systematization of a set of principles that were not issued until then. In this way, free software paved the way for the modern model of networked software development based on the principles embodied by the four freedoms of free software, providing, with the GNU GPL, "a kind of written constitution for the hacker world that enshrined basic assumptions about how their community should function" [Moody, 2002, 27].

[37]From the perspective of the text, this ethos was visible in the reuse of texts snippets in Stallman's communications and in the precedent of the General Public License, represented in the Emacs license.

4 — The open network paradigm

4.1 Introduction

The influence of computer networks encompasses almost every aspect of life –directly or indirectly– just consider the first two decades of the 21st century. Although this statement may sound somewhat obvious, given the current state of technology, it bespeaks of the profound change that software, particularly collaboration in the development of modern software, has undergone based on the technological and discursive arrangement represented by networks. This chapter will explore the relationship between open collaboration in the software culture and networks under the framework of the so-called *networked society*[1]. But first, to clear the way for the argument, some considerations are necessary regarding how this relationship between software and networks is assumed. On the one hand, and considering the notable effect of networks on the stages of software creation and distribution, reducing the exchange between software and networks to merely an influence would be misleading, pointing to a kind of network-based techno-determinism. That modern software development practices were significantly improved by the use of networks is undeniable, but it would be contradictory with the vision of technology that I have sustained in the assessment of networks –particularly computer networks– to adjudicate it a merely instrumental role. If under the originary technicity construct and through an inference operation, software was seen as a technology (although a unique one), inheriting all the complexity of technology, an analogous operation can and must be applied

[1]Here, I follow the terminology of Manuel Castells where, in short, the network society is characterized by "the preeminence of social morphology over social action" [Castells, 1996, 469] networks being the "new social morphology of our societies" [Castells, 1996, 469]. In other words, networks have increasingly become one of the organizing principles of this society, its topology-morphology serving as a metaphor to describe a wide range of economic and socio-cultural processes. Although I subscribe to this view, some adjustments will be needed to establish a fruitful dialog with this definition. It will therefore be mentioned throughout this chapter.

to networks. That is that networks are not mere tools and are subject to other social and ontological connotations.

Furthermore, and as it will be shown, the history of networks shares many threads, motives, and protagonists with the history of software, as depicted in the last chapter. Then, on the other hand, and according to the last observation, the boundaries between networks and software can be described as blurred. In previous chapters, I mentioned that a software-hardware tandem is a useful abstraction to understand the multiple negotiations surrounding the materiality of code without losing sight of the intertwined nature of the software-hardware assemblage, in which, although software and hardware are different, they cannot be considered separately as a symbolic layer over a physical layer. A type of software-network tandem can also be termed, for the sake of simplicity, always stressing the abstract nature of this representation that obeys software (and its open, collaborative practices) as an organizing factor. Actually, the conception of computer networks acknowledges them as a thin fabric woven from software, hardware, and telecommunications technologies along with social structures that together shape a discursive realm that changed how programming was practiced. This last observation hints again at the complex nature of networks and calls for a methodological approach that addresses this premise. For this reason, the Actor-Network Theory (ANT) has been selected.

Although some particularities of ANT will be developed in the next sections, it may be useful to outline why this approach was selected and how its viewpoint is interpreted, mentioning which concepts apply specifically. I am well aware that there is some criticism of ANT, given its yielding structure and the generality of some of its constructs. However, it is precisely this flexible nature that gives the desired adaptability to my purpose, reflecting in the process the fluid nature of networks. The idea is to draw some concepts from the theoretical corpus of ANT that operate as agents applied to technical developments and milestones without the constraints of a rigid superstructure. In this way, a hybrid narrative is built between technical reports and a critical perspective on networks under the lenses of open collaboration. This wording in no way implies ANT's lack of a general axis or the inability to make claims. To illustrate some of these general lines, consider Castells and Latour's assessment of networks, the latter being the central figure of the actor-network theory. While Castells, when explaining his idea of network society, succinctly characterizes network as a "set of interconnected nodes" [Castells, 1996, 470], perhaps relying too much on a topological and

infrastructure perspective, Latour characterizes a network (or translation)[2] as "[m]ore supple than the notion of system, more historical than the notion of structure, more empirical than the notion of complexity, the idea of network is the Ariadne's thread of these interwoven stories" [Latour, 1993, 3], a kind of definition by comparison where the fluidity of the object is clearly established. An opposition between the two positions (although both are contemporary, they have incomparable goals) cannot be drawn, they only acknowledge ANT's malleability, again explaining its convenience and presenting some recurring topics, such as *system* and *history*.

After this explanation, a range of observations unfolds supporting the decision of using ANT. First, the tension between the notions of system and network itself marks a paradigm transition that has historical roots and that, at the same time, highlights software's shift from a calculation process to a communication medium. Second, the notion of naked abstract complexity is superseded by the assemblage conveyed by the collective, which, even if complex, remains more empirical by observing the interactions between actors (or "actants", following the terminology) represented by human and non-human entities. Thirdly, because ANT considers networks open-ended processes, continually undergoing change and evolution, it calls for a historical account rather than a static vision [Krieger and Belliger, 2014, 91]. Finally, the entire theoretical construction of ANT, as well as Latour, besides proposing a historical assessment of networks, also question the dualities often posed by the teleological rationality that marks so-called modernity. By questioning science and technology, using the evolution of our understanding of these concepts, ANT brings forth the problems and some mutual exclusions, such as object/subject, nature/society, and so forth, that are historically rooted in the now controversial vision of enlightenment. Although the context is different, the oppositions that I have already mentioned, such as knowledge/technology or more so hardware/software, can also be questioned –methodologically speaking– through ANT. Furthermore, if Latour's statement that "technology is society" resonates in some way with the holistic position of the originary technicity, then it should be admitted that such binary oppositions are less suitable for network analysis than the full recognition of the complexity and the previously mentioned hybrid nature of networks. This position requires multidisciplinarity and also the admission of networks as a socio-technical complex, which differs from an instrumental

[2]Although the term Translation is too complex to address here, it can be said that it accounts for the complexities between the social and the technological, the artifacts, and the diversity of different actors' objectives in solving a particular technological problem. See [Callon, 1984]

opposition between users and a topologically shaped network infrastructure.

Perhaps one of ANT's problems is the ambitious all-encompassing scope of its approach. Therefore, we must keep in mind that given the interest in software and collaboration, the discussion will revolve around computer networks and how the software culture influenced their development and how, once established, the network logic became entangled with the software logic. As previously mentioned, the historical account is of paramount importance in the ANT practice; therefore, this chapter follows the vein of the previous chapter and outlines the history of networking. Obviously, such an undertaking, for reasons of space and argument, has to address a small set of accounts, selected to support a determined claim. This chapter is organized around the following sections. As in the previous chapter, the first section presents a prehistory of networks that sheds light on some well-known twentieth-century precedents, in which the devices envisioned followed rather pragmatic purposes. The next section deals with Arpanet, supposedly the first successful computer network that is simultaneously an example of collaboration in software culture and a reference point from which the foundations of the network culture were established for the future. A fourth section reinforces the connection with the previous chapter by reintroducing the Unix operating system and its associated hacker culture from a network perspective in which this system played a major role in the standardization of network software. The following section also introduces USENET as an example of a Unix-based network developed in a grassroots personal computer environment to see the logic of software collaboration outside of Arpanet's rigid circle. Once this historical background is established, the next section will have a more theoretical tone, analyzing the protocol as a socio-technical concept built within the collective as a response to something that I have termed, in clear parallel with the software culture, "the network crisis", providing comparable causes and effects. The consolidation of modern open collaboration practices in software culture is described in the two final sections. One will address the emergence of the Open Source movement, which is considered –following the previous chapter– as the third iteration on the hacker culture. The second will further discuss some possibilities of the open-source approach to software development under the metaphor, coined by Eric S. Raymond, of *the bazaar* [Raymond, 2001]. The conclusion will provide a critical discussion on the development of software collaboration in the form of networks, presenting it not as a deterministic consequence of the emergence of the Internet but as a very constitutive element of the network society. Before proceeding, two clarifications should be made.

On the one hand, and in order to articulate the historical with the theoretical, I build upon my interpretation of concepts such as "social operating system" and "distributed agency" as included in the reading of ANT by Krieger and Belliger [Krieger and Belliger, 2014]. On the other, to reflect the nature of networks in the text, the chapter has been written in simple sections without any subsections or further subdivisions. In this way, it differs from the nested structure of the previous chapter, conveying somewhat independent (yet connected) narrative nodes.

4.2 Node 1: networks foreseen

It is almost common to trace the origins of computer networks back to some now seminal documents that go back to a time when computer technologies were still in their infancy. And although this tendency correctly points to the fact that ideas pertaining to some of the premises of the networking effort were in the air, the procedure must be carefully framed, taking into account the precise historical moment in which these ideas were planned. In other words, these contributions must be seen in perspective, more like visionary recommendations than a blueprint for a technology that did not exist, but whose possibilities were desirable. Specifically, a set of well-known works by Vannevar Bush and J.CR. Licklider will be discussed, taking into account certain aspects. The first one is the sources discussed; for instance, the famous 1945 magazine article "As we may think" by Vannevar Bush [Bush, 1945] or an ARPA internal memorandum dating back from 1963 entitled "Memorandum for Members and Affiliates of the Intergalactic Computer Network" by J.C.R Licklider [Licklider, 1963].

Neither of these texts was scholarly papers; thus, it could be argued that precisely because they were outside of academic constraints, their authors –academics– indulged in a more speculative vein. Secondly, and related to the previous, is that most contributions were penned while their authors held important positions within the government related to technology policies; this is noteworthy because Bush and Licklider were not simply prospective academic thinkers, they were also decision-makers who had the potential to finance key experimental undertakings. Moreover, their inclination to fund academic research would shape the way computer networks were conceived and deployed. The third aspect is the window of time between the contributions of both authors. When Vannevar Bush conceived his "Memex" and made his predictions about how information should be managed, electronic computers, let alone "stored-program computers", were mainly on the hori-

zon; this explains the preference for analog technologies in his vision. Almost twenty years later, when the software culture was already being nurtured in corporations and universities, Licklider gave his written proposals a more down to earth tone, detailing technological requirements in the form of languages and protocols and, indeed, referring specifically to networks. That said, the purpose of this exercise goes beyond the simple historical framing that legitimates possible gaps in the computer networks timeline with precedents and the brief focus in the paradigm shift from mainly systemic thinking to one of a network. If network thinking, predominantly influenced by computer networks, became a metaphor, then looking at this paradigm shift reveals some qualities of the software-networks assemblage.

Simply put, Bush's words can be analyzed from a cybernetic point of view, whereas Licklider already exhibited a kind of proto-network thinking. The previous leads us to the following question, was there indeed a paradigm shift, or are systems and networks simply different terms to refer to the same thing? Although this question will be addressed later, it is important to note that up to now, a more systemic approach has been applied to software and that the change of the computer from a calculation machine to a medium permeates our understanding of this question.

In retrospect, Vannevar Bush's insight on the way "we may think" allows us to channel (in 1945) some technologies to come against a backdrop of industrialism and massive production. In describing his devices and their information management techniques, Bush offers a now historical perspective in which the growing complexity of science demanded not only new technologies but also new ways of storing, classifying, and accessing the also increasing amounts of correlated information. Although it would be misleading to state that Bush envisioned a network in his proposal, some of the elements in the network paradigm were present[3]. For starters, it can be said that the primary motivation behind Bush's writing is the science enterprise. As mentioned by Wardrip-Fruin, the Memex (the main device that will be briefly described) was conceived "as means of turning an *information explosion* into a *knowledge explosion*" (emphasis in original) [Wardrip-Fruin and Montfort, 2003a, 35], pointing to the high volume of information relevant to modern science. Bush characterizes science as a communicative process between individuals,

[3]To clarify, it should be noted that new media literature often presents Bush's seminal article as a precursor to hypertext or even of the World Wide Web, rather than computer networks. In fact, the main invention of Bush's thought experiment –the memex– is still an electromechanical device and there is no indication of direct connectivity between the different instances.

where the recording of ideas and their access is a condition of the evolution of knowledge, as described in the previous chapter in which software openness was traced back to the method of science. Therefore, the existence of a "growing mountain of research" makes not only scientific work more difficult and specialization necessary because of the vast amount of information but also requires new approaches to information management [Bush, 1945].

From my perspective, this is Bush's twofold contribution; first, proposing an arrangement of devices to provide a better way to handle different records and second, a new logical system of storing and retrieving these records. In the first case, technologies such as microfilm, photography, and voice to text conversion, among others, were a set of diverse technologies that provided increased record evaluation. Although Bush does not discuss specific electronic computer technologies and supports his arguments on electromechanical inventions –something that now seems outdated– he does recognize the possibilities of information when it encompasses various media. In some way, this precedes the software-driven electronic computer and its transition from a calculating machine to a media machine and reinforces the communicative nature of the scientific endeavors. Secondly, and as a complement, a new information management approach was needed to profit from the "described devices" advantages and, especially, of a new generation of machines that were not (like the others) merely an extrapolation of a particular technology, the Memex. The Memex consisted of a desktop with two main displays, optic notepad, and an inner microfilm-based mechanism, as well as levers that allowed the user to make connections or "trails", linking several records related to a particular path of reasoning in a non-linear way. These connections could, in turn, be archived and retrieved as needed. This proposal creates a point of contention between the nature of information and the technologies that surround it; namely, that mass production facilitated new inventions but failed to provide innovative means for information manipulation. This new approach was a system that was better suited for information management than the well-established hierarchical approach of nested subclasses. In Bush's words, this system should function more like the human mind in which ideas follow association, not only indexation. Thus, the "associative index" [Bush, 1945], which was materialized in the Memex whose, as mentioned before, trails linked several sources following the researcher's particular intention.

This provocative proposition, the associative index, established Bush as a pioneer of hypertext; and, although his vision was far removed from current technologies, his description of logical systems (more than devices) has

remained a breakthrough in information processing. Although Bush was thinking in more of a library than a computer –let alone software or a computer network– the qualitative leap of his ideas was to break the linearity associated with this top-down hierarchical information processing and to embrace the fragmentation embodied by his trail concept. In doing so, media could be re-arranged in different ways, and the resulting trails could circulate, enabling a type of information sharing system for scientific purposes. As Bush states, "[t]he (the memex user) sets the reproducer in action, photographs the whole trail out, and passes it to his friend for insertion in his own memex, there to be linked into the more general trails" [Bush, 1945, 46].

The conclusion is twofold. Bush's ideas recognize the multimedia nature of information and make more visible the communicative trail of science, which requires new paradigms of information management. Perhaps this signals bringing into the discussion –although briefly– some fundamental elements of cybernetic ideas. It is also noteworthy that Bush's proposal was (like cybernetics) a post-war inquiry that was mainly a mesh between machines and users' information. It could be said that early cybernetics was a shift from rational, mechanistic thinking in which the machine was an isolated entity, to a more systemic paradigm based on "communication and control systems that spread across multiple entities" [Wardrip-Fruin and Montfort, 2003c, 65]. However, this should not simply entail a reduction; as Wiener expressed, "a human interpretation of the machine, or a mechanical interpretation of the operator, or both"[Wiener, 2003, 68]. It should be a broader perception in which the concept of systems –not just the sum of the parts– played an important role. Finally, and to conclude, the last comment represents Bush's starting point of cybernetics. While associating an entire theory with merely the speculative statement as "As we may think" is clearly an asymmetry, the key point here is to address the distinctive goal of each approach. Both of them acknowledge the existence of a close man-machine relationship. However, while cybernetics is more interested in the control of the combined gun-operator assemblage, Bush was thinking ahead to the information explosion and ventured a possible solution through his memex-user assemblage. Furthermore, the abstraction of information embodied by the trail concept and its combining and flowing features could be compared to a theoretical construct such as packet switching, whose implementation is one of the principles of computer networks.

The path foreseen by J.C.R Licklider is allegedly the clearest antecedent of early network thought framed precisely in the era of the electronic computer and software. Through a series of writings, Licklider represents the inter-

section between cybernetics applied to computers ("Man-Computer Symbiosis"), information management, echoing Bush's concerns ("Libraries of the Future"), the computer as a medium ("Computer as a Communication Device"), and explicitly, a call to action in computer networks implementation ("Memorandum for Members and Affiliates of the Intergalactic Computer Network"). As mentioned, Licklider was in a position of privilege. His appointment in the recently created ARPA allowed him to work directly with computing technologies and fund, as well as inspire, groundbreaking research in the field. Take, for example, time-sharing systems already discussed. Not only did these systems arise, in part, thanks to Licklider's administrative support, but they reflected the kind of pragmatism that also drove the computer network effort [Brate, 2002, 92]. To summarize, if time-sharing was developed to improve the cost-effectiveness of expensive computers by allowing different users to use them simultaneously, networks offered a similar opportunity by combining the computer resources (including computer programs) scattered in different computer installations. Licklider is certainly one of the champions of the revolutionary idea that computers were far more than number-crushers. It was noted that he wanted "to move computing research away from batch processing, away from the idea of the computer as arithmetic engine, and toward the computer as a communication device"[Brate, 2002, 94]. To explain his contributions to computer networks, I will first address some ideas included in his "Man-Computer Symbiosis" and the "memorandum" to delve deeper into the discussed paradigm shift from cybernetics to networks. However, I am not implying a correspondence between these paradigms and the writings (as their title might suggest) because the two writings draw from both paradigms.

First, it is clear that for Licklider, the so-called symbiosis between man and computer was a subclass of the more general man-machine systems [Licklider, 1960, 4] that viewed the symbiosis as an expected development in the cooperative interaction between both parts (man and electronic computers) [Wardrip-Fruin and Montfort, 2003b, 73,74]. While Bush blamed industrialism in part for not providing an adequate solution to information overload, Licklifer did something similar by observing that although there were several man-machine systems, man-computer systems were still unsatisfactory and unable to provide an adequate symbiosis [Licklider, 1960, 4]. Licklider's call for a symbiosis presents the man-computer assembly as a hybrid that has not yet been properly exploited and exposes the strict dividing of the tasks to be performed by each part of the hybrid as a still systemic model. However, a departure from the fixed structure of systematic thought is first achieved by symbiosis being recognized as *flexible*, evidenced in the partiality

towards a cooperation between the hybrid parts independent from predetermined programs [Licklider, 1960, 4] or the recommendation of an intuitive trial-and-error procedure "in which the computer cooperated, turning up flaws in the reasoning or revealing unexpected turns in the solution" [Licklider, 1960, 5]. These specific observations could be assumed as a response to the determinism of programming conceived as a linear practice and point to its inadequacy in solving problems in a cooperative man-computer fashion, as also observed in Licklider's promotion of interactive computing against batch processing. These comments are also close to the foundations of proper proto-network thinking in which distributed artifacts, and their agency could offer a powerful realization of symbiosis (taking into account the pragmatic rationale). Consider these two examples. Licklider evokes time-sharing systems as a development to procure a more cost-effective use of the computer (granting access to several users), and, from that, by analogy, he envisages a kind of "thinking center". This hybrid between the library and the information processing facility's interconnection is clearly described as "a network of such centers, connected to one another by wide-band communication lines and to individual users by leased-wire services" [Licklider, 1960, 7]. The second example addresses not a topological issue but a more abstract issue and, perhaps, one that is more compatible with the described infrastructure and the trial and error approach. It pertains to the instruction of computers based on "real-time concatenation of preprogrammed segments and closed subroutines which the human operator can designate and call into action by name" [Licklider, 1960, 9].

Although not focused on networks, the search for a true symbiosis between man and computer already raises networking concerns that would be fully expanded in the famous "memorandum". This document (by definition, an internal ARPA document) presents other pragmatic and technical issues in network development. Regardless of what one might infer from such an exotic and vague adjective as *"galactic"*, the writings, dating back to 1963, use a hypothetical case that illustrates the potential benefits of networks. As mentioned, Licklider's appointment to ARPA allowed him a privileged view to assess the increasing complexity of actors, projects, resources, and technologies –a "collective" in ANT terms– that were involved in developing new information technologies, improving intellectual capabilities (through human-computer symbiosis), and providing new approaches to the theory of science [Licklider, 1963, 1]. Licklider observed that "[t]o make progress, each of the active research needs a software base and a hardware facility more complex and more extensive than he, himself, can create in reasonable time" [Licklider, 1963, 1]. This statement points out the insufficiency of individual

efforts in obtaining results and the convenience of a networked assemblage in obtaining them. Thus, the memorandum's main contribution is considering this extended software/hardware scenario as the framework for a mental experiment in which a user engaged in a scientific task requests the services provided by a network of a different system. In this scenario, the imaginary user looked for local programs to help him with his purposes and, when no alternative was available, he navigated the network to find the proper program. In doing so, the issue of interoperability between programs stands out as one of the possibilities of the ensuing network. Moreover, because the user sometimes had to produce a new source code for an unfulfilled requirement, a kind of online link to other code located elsewhere was proposed [Licklider, 1963, 4,5]. Thus, it can be stated that Licklider anticipated –although rudimentarily– inter-program communication and the use of code modules. More importantly, his discussions exposed the issue of protocol. His vision featured several key network operations, namely, program retrieval, linking, as well as the borrowing and storing datasets of the entire process for others to use [Licklider, 1963, 6]. However, in describing the flow of these operations, he also noted the complexity of the negotiations between the parts. For example, while a common "blackboard" was something desirable to assess mutual dependencies between participants, it was advisable not to impose strong restrictions on each system to indicate in the decision process what decisions were to be made by the overall network or by an individual participating system.

To conclude, although different in aim and tone, both of Licklider's writings are comparable. They can be seen as complementary because the network envisioned in the memorandum was likely composed of interactive systems, that is, time-sharing systems, which, in turn, were an example of the idea of human-computer symbiosis. The two writings also suggested the fragmentation of programs as a way of dealing with complexity. Moreover, although the language used indicates that, for pragmatic reasons, systems and networks are similar, the memorandum specifically points out that while networks could solve several problems, they also resolved the problem of insufficiency of resources (programs and hardware) through the coordination between the nodes.

Finally, and regarding the contributions of Bush and Licklider, several aspects can be noted from this background of the history of the network. Like the argument presented in the previous chapter, where the practices of collaboration and sharing in the history of software can be traced back to some lines in the paradigm of science, in early network thinking, science also had a

prominent place. This idea is understandable considering that, at the time, computers were widely used in scientific calculations. This observation introduces us to the subtle critique concerning the shift from a systemic paradigm to networks. Simply put, that the systemic paradigm was more compatible with the rationalist model of science, while the network paradigm resonated better with the questioning of that model. Consider Bush's proposal and his Memex. His criticism of hierarchical information management based on index and subclasses in favor of "associative indexing" points directly to the encyclopedia as the artifact par excellence of information management of rationalist enlightenment. While the roles of the editor and reader were well defined in the encyclopedia, in the Memex, it was the user who was in charge of organizing the information and sharing it with other users, *"jeder benutzer soll zum Herausgeber seiner eigenen, associativen Pfade' werden"* [Wirth, 2005, 50]. Looking at this paradigm shift from the ANT perspective, it can be said that the general idea of networks is one of the metaphors used by Latour to describe the hybrid arrangements in scientific practice more precisely, as well as the complex structure that contests dichotomies. It can also be argued that this alternative (when compared to system) highlights two qualities of the network, flexibility, and communication purpose. By questioning one of the basics of the rationalist scientific paradigm, namely, the division between the subject (the scientist) and the object (the matter of inquiry), ANT reconfigured the relationship of the active human, observer of the passive reality, into an arrangement of actors (or actants) whose negotiations represent the scientific enterprise and include specific cultural and social elements as part of it.

From an admittedly loose reading of the issue, parallels can be drawn between this new role of the researcher as part of an array and the inclusion of the observer as part of a system in second-order cybernetics. However, while systems may address some of the characteristics of these emerging ideas, they are more suited, as previously noted, to representing pre-planned hierarchically ordered partitions (slightly flexible) and well-defined communications. Networks, on the other hand, offered openness through a third quality, a more dynamic metaphor where change was always expected. In short, systems could also model the ongoing negotiation needed by ANT, but because they tended to be closed and their modification painstaking, networks represented a better descriptor. Because the network structure is based on interconnected nodes, which allows for the addition of new nodes (the architecture of networks can be designed to be open), they embodied the desired flexibility where communication could take place following multiple paths. As a result, they represent the metaphor of choice to model

from an epistemic critique to a computer arrangement for sharing resources. However, a final point should be made. Thinking of networks from a merely topological perspective implies an incomplete and even instrumental picture. As technology, networks can be considered a part of originary technicity, and their vocation for communication only reaffirms this position. By definition, communication was fundamental to software collaboration. By looking at the computer as a communication device, as Licklider and Robert Taylor (one of the designers of the Arpanet) stated, a new age was on the horizon in which communication between people via the machine would be easy, allowing new ideas to emerge [Licklider and Taylor, 1968, 21].

4.3 Node 2: Arpanet and the development of the network paradigm

ARPANET was the first and supposedly the most influential technological achievement of the computer network concept. Its implementation proved not only various technical ideas but also established many features of the network culture, which are currently taken for granted. The advent of ARPANET directly preceded –technically and socially– the currently ubiquitous Internet. In turn, its structure and building process reveal a complex arrangement of actors, protocols, and devices interacting collectively [4]. With that in mind, the ARPANET will be addressed both historically, as a technology that set the foundation for modern software collaboration and also as an instance of the ANT theory's more general construct of the network. As the first computer network, ARPANET surpassed its initial instrumental and pragmatic nature, and, in some aspects, displayed some collaborative features. Additionally, the complexity of the undertaking required a design in which communication should play a key role, posing some problematic issues when dealing with the intuitive collaboration-communication duple of the ANT perspective. However, this field of tension enabled the dialogue between ANT and other notions about collaboration. But first, the difference between the terms of system and network must be settled in favor of the latter. From the interchangeability applied to both terms in some accounts, one might assume that they are equivalent. Take, for instance, Licklider's phrasing in devising his famous galactic network. After acknowledging the complex combination of languages, persons, hardware, and software, etc. contained in his proposal, he comes up with a *desideratum* of the ideal fea-

[4] Here, I follow my reading of the ANT theory in which the collective can be understood as a composition between the "community" and non-human actors in a network.

tures for this "group, system, or network", regarding the three terms as the same [Licklider, 1963, 2]. Concerning technical documents, one of the early descriptions of the Interface Message Processor (IMP), a key element of the origin of the ARPANET and forerunner of the modern router, deems the subnet produced by the IMP a *communication system*, making it possible to reduce the notion of networks into a systems rationale. However, this can be problematic given ANTs non-reduction principle; nonetheless, it follows one of the basic principles of the systems theory, namely, that everything can be interpreted as a system [Krieger and Belliger, 2014, 69].

Although it can be said that every network is a system, considering the implication in the opposite direction seems misguiding, to say the least. Moreover, although both systems and networks deal with complexity, they perceive it in different registers. In systems, it is considered a problem to be solved by organization; in networks, although also complex, they "do not arise from the need to reduce complexity" [Krieger and Belliger, 2014, 71]. Based on this constitutive difference, I sustain that the idea of networks is better to describe modern open software collaboration than systems. Moreover, while admitting that the two terms sometimes overlap, the preference of networks over systems obeys some structural reasons that I call the historical, architectural, and communicative principles of networks. First, historically, the end of the 1960s marked a milestone in the emergence of the network metaphor within the computer and software culture. The narrative of the software crisis was accompanied by technical innovations (structured programming) and economic disputes (IBM's unbundling process), which are clearly a product of the issue of complexity in the software realm, and therefore more system-oriented. This complexity offered software a new potentiality resulting from communication issues that made insufficient the existing conception of the computer as a calculation machine. Thus, a term already in use in the field of media (e.g., radio network) became a suitable choice to classify this new computer, evolved into a media machine, and its possible connection to other peers.

From an architectural perspective, the open nature of the ARPANET design, as the original computer network, not only set the common denominator for other future undertakings –notably the Internet– but also highlighted the flexibility of network tropes compared to the system's metaphor. In other words, the ability of networks to incorporate new nodes in a continuous process was unparalleled in the definitions of system, which were more oriented towards defining a system as a closed entity composed of well-known predesigned parts. This is not to say that a system cannot be improved with

new elements. However, its emphasis on control and solving a particular task do not easily explain the chaotic building of open network architecture. Meanwhile, the communication aspect of networks points to another difference. Discussing this particular element is difficult because, although fundamental, neither the nascent computer sciences of the '60s or the later socio-technical theories, such as ANT, were well equipped to engage with it [5]. For now, however, suffice it to say that the breakthrough of networks was the packet communication. If the linearity of early software was progressively broken by jumps, subroutines, modules, etc. to manage complexity, packet message networks worked a similar process on the linearity of the classical transmission metaphor. Thus, although systems already included communications and even the cybernetic discourse recognizes its importance, systems, in general, are about control and are also governed by the image of linear transmission, while networks adopt fragmentation. For these reasons and returning to the main issue, ARPANET is an outstanding example of the notion of the computer network and was instrumental in making collaboration a metaphor.

ARPANET was born mainly out of a pragmatic need to connect existing computers to optimize software and hardware use, given the lack of a particular resource in local installations. In other words, the complexity resulting from the multiplicity of platforms led to the issue of incompatibility, that is, programs that only ran in one location although they could be useful to others [Marill and Roberts, 1966, 425], motivating a network without standardization enforcements (as Licklider correctly foresaw) [Marill and Roberts, 1966, 428]. As later Larry Roberts, one of ARPANET's main advocates, recalled, "I concluded that the most important problem in the computer field before us at that time was computer networking; the ability to access one computer easily and economically to permit resource sharing" [Roberts, 1988, 143]. In fact, ARPANET's original announcement, dating from 1967, lists the network's possibilities as load sharing, interpersonal communication, data sharing, program sharing, remote service (remote program execution), specialized hardware access, specialized systems software use and scientific communication[6] [Roberts, 1967, 1].

[5] As Farias noted, ANT has a shortcoming when dealing with communication, given its empiricism and its real nature, where communication is mainly overlooked. However, this gap is an opportunity to expand ANT [Farías, 2014].

[6] Here, it is important to highlight that personal communication is an advantage of networks, given that although it was considered, it was deemed as unimportant. This supports the idea of a network based on an open architecture, whose final use and evolution cannot be wholly controlled or planned.

Not unlike other computer advancements in that era, ARPANET was the product of an academic environment funded by the Department of Defense of the United States, with the support of a private company (BBN) with strong ties to the academic community, and some universities. To be more precise, both the already mentioned ARPA (Advanced Research Projects Agency) and its IPTO (Information Processing Techniques Office) were interested in exploring the possibility of sharing computing resources by connecting the different installations of the projects being executed with the support of the agency (mainly in universities). Network folklore tells how the then IPTO director, and Licklider's successor in that position, Bob Taylor, had three different (incompatible between them) terminals in his office connected to time-sharing systems at MIT, Berkley, and SDC (Systems Development Company – California). He noted that although each system had developed a community, they operated in isolation from each other [Hafner and Lyon, 1994, 13]. Beyond the anecdote, this point is significant because it establishes a link between time-sharing systems and networks research, where ARPANET could be seen as a *network of systems* with obvious collaborative potential. As noted by Thomas Marrill and Lawrence Roberts, "we can envision the possibility of the various time-shared computers communicating directly with one another, as well as with their users, so as to cooperate on the solution of problems" [Marill and Roberts, 1966, 426].

It was Lawrence (Larry) Roberts, one of the authors of this last quote, that Bob Taylor readied to be the manager of the ARPANET, managing the assemblage of several technical innovations. He became the person in charge of this complex socio-technical network[7] in the making, which included universities, companies, hardware, software, documentation, communications infrastructure, and others. Roberts had already experimented with networks by connecting a TX-2 computer in Massachusetts to a Q-32 in California using a low-speed dial-up telephone line in 1965 [Leiner et al., 2009, 23]. This direct precursor of ARPANET not only proved the feasibility of connecting two systems but also showed the drawbacks of the regular telephone network for carrying computer communications. This finding was crucial in deciding to adopt a revolutionary paradigm such as packet communication as the option to achieve the desired purpose. Packet communication was a theoretical concept developed simultaneously by MIT, the RAND Corporation, and the NPL (National Physics Laboratory – England) in the first half of the six-

[7]From here on, I will use the term socio-technical network instead of "system" to stress the convenience of the network metaphor, whether technical or conceptual.

ties [Leiner et al., 2009, 23]. It was clearly a technological challenge, given its radical approach of dividing communication into fragments (packets) that were routed following different paths and could subsequently be re-assembled by the receptor. At this point, the IMP (Interface Message Processor) comes into play. Developed by Bolt Barak and Newman (BBN), following a contract with ARPA, it was itself a complex assemblage of hardware, software, and private companies. The IMP embodied the abstraction that made the network work, eliminating the potential technical overhead of complexity. A sub-network of interconnected IMPs would manage the packet communication based on a common protocol, and each party of the time-sharing system would only have to figure out how to connect to a one IMP and not to all of the other (incompatible) systems. As history would show, this intelligent design decision would initiate the entire network culture. The ARPANET made its first successful communication in 1969; it became completely operative (enough to make a public demonstration) in 1972 [Hafner and Lyon, 1994, 153,181] and served as the blueprint for future network achievements.

ARPANET's importance to open software collaboration can be assessed at various levels. The most obvious is that it was an underlying technical layer that facilitated communications between cooperating parties. While this is correct, strictly perceiving computer networks this way reduces the network to a means to an end, that is, is a tool. This view may coincide with accounts that advocate the vision of network topology, but fail to see the computer network as a collaborative effort in itself. This topological constraint must be overcome in favor of finding the connection between ARPANET's open architecture and some open practices present in its development. It should be noted, however, that, as part of a socio-technical network, these practices have different degrees of openness and are scattered across the components of the network. To that end, and considering ANT's pragmatic approach, I have selected three attractors to observe some instances of collaboration and/or openness in socio-material practices (software, hardware, community); the latter will be addressed in-depth in the following section on protocols. In doing so, not only will ARPANET's technical and empirical evidence be asserted, but some of the artifacts and behaviors inherited by the Internet will also be highlighted.

- Software: ARPANET's software development can be roughly divided into two groups. The first group is the software built for the IMP, which routed the packets to other IMPs and offered a standard connection facility to a local time-sharing system. The second is the systems software that granted access to the attached IMP while maintaining the

idiosyncrasies of each system (their own operating system, programming language, hardware features, etc.). Because the latter varied in each installation, its description would be challenging; therefore, we will examine the IMP software briefly. Moreover, the real innovation was the IMP software, which had to implement the services of uncharted territory, as was the packet-based computer communication. The IMP software group was small. It featured three members who worked for a year and knew all the code [Heart et al., 1970, 559] and, therefore, could write code or rewrite someone else's work [Hafner and Lyon, 1994, 145]. Thus, "all three members knew the entire software system inside out" [Walden and Guys, 2014, 28] although each was in charge of a part of the system, namely, tool development and statistical functions, interaction between IMPs, and the connection of the IMP with the host computer. This collaborative workflow was made possible because the IMP code was developed using another *network* of systems, including a PD1-1 time-sharing system to edit and compose the programs, a DDP 516 (on which the IMP was based) to assemble, and the SDS 940 printer to code listings [Heart et al., 1970, 564,565], which granted code availability to the developers. Once completed, the system was loaded into the actual IMP using paper tape (the upgrade was later performed using the same network connection, prior to the automatic software upgrade, common today) [Walden and Guys, 2014, 29]. Because IMP software delved into the unknown complexity of implementing packet-based communication and almost seamless 24/7 service, it can be argued that new approaches, such as conveniently structured programming helped the process, as the structure of "many nearly self-contained subprograms" suggest [Heart et al., 1970, 557]. Ultimately, the IMP code was reused, reimplemented, extended, and changed as it informed the implementation of other networks and, in general, the Internet [Walden and Guys, 2014, 28], which, in turn, shows the potential for open architecture design and the agency of open and distributed code[8].

- Hardware: Not unlike software, the hardware layout consisted mainly of two groups, the hosts and the IMP. The first term was coined in

[8]This affirmation does not imply that IMP code was open or free by current standards. Because it was developed in house by BBN to comply with a government contract, the company probably had strict control of the code base. Considering that at the time software had no clear commercial value and the multiple parts of the socio-technical network required to accomplish projects, such as ARPANET, it is understandable that the code flowed through different artifacts, encompassing code listings, paper tapes or the computer network itself.

the project to refer to the different types of computers with a time-sharing system connected to the ARPANET through their respective IMP, which, in turn, consisted of a small adapted minicomputer (and associated software) that managed host to host communication by routing data packets back and forth between them. Properly speaking, the IMP was a robust version of the Honeywell DDP – 516 [Hafner and Lyon, 1994, 97], which was modified by BBN engineers to conform to the specifications established by the project management at ARPA. In a move reminiscent of the hardware wiring performed by McCarthy to achieve multitasking, the 516 had to be adjusted to allow for low-priority interruptions [Heart et al., 1970, 562]. This observation points to an essential feature of the entire ARPANET venture, and that was the collaboration between the project and the DDP 516 manufacturer. Because there was no previous experience in custom hardware for packet-switching and several IMP units were expected, BBN engineers had to experiment in the manufacture of the hardware, sending design modifications to Honeywell to have them produced in series; a process that was not free of misunderstandings and delays [Hafner and Lyon, 1994, 125]. Besides the modification to make program interruption simpler, other hardware changes were made, mainly the design of interfaces to support inter-IMP communication with the modem (IMP/Modem) and with the Host (Host/IMP) [Walden and Guys, 2014, 28,29]. Each interface had specifically associated complexities and the feasibility required to allow full-duplex communication. For instance, while the IMP/Modem established a connection using existing telephone lines to route switched packet communication; the Host/IMP interface had to cope with the diversity of existing platforms, for which the IMP had a standard interface and every Host was responsible for designing its own interface, following a set of rules [Bolt and Inc., 1969, 34,35]. Last but not least, the hardware was also subject to iterative evolution, [Bolt and Inc., 1976]. An updated version of the IMP was completed for the 316 minicomputers, based on a cheaper version compatible with the 516, as well as a new device called the "terminal IMP", which met the growing demand by allowing a connecting host and 64 additional terminals, which did not require their own host[9] [Bolt and Inc., 1976, 1-1].

- Community: Although the two previous examples focus on the tech-

[9]Taken from an updated version of the original BBN report number 1822 (1969), which described the Host/IMP communication in detail. In the absence of another name, this protocol was named Protocol 1822.

nical side, they define the role of the community as part of the socio-technical network. These examples also show the preponderance of communication at different levels (between companies or volunteers) to coordinate tasks that aim at a shared objective. Despite ANT's problematic reading of communication, the relationship between community and communication is evident. However, the issue is more complex, as simple common etymology might suggest. For now, suffice to say that ARPANET gave birth to a new community made up of students and early network enthusiasts, whose volunteer work determined the style of how network standards were constructed. Unlike other ARPANET related communities, such as researchers at ARPA or employees at BBN, the Network Working Group (NWG) was heterogeneous and created more organically out of the needs left by the gaps in the early specifications of the network. The BBN guaranteed that the IMP would deliver data packets from host to host, but what would happen after that was not yet clear or even designed. Here, the group –not yet called NWG– entered the picture to propose how real host to host communication would be achieved and how requests for resource sharing or data transfer would be accomplished [Hafner and Lyon, 1994, 145]. Thus, the NWG produced two important outcomes, the Network Control Protocol (NCP) and the RFCs. Considering that the latter is still used, I focus on the role of open documentation as an expression of the community, highlighting the NCP as part of the communication between machine actants.

The ARPANET's open network design, involving a heterogeneous set of parties, and its complexity made the internal communication process required to achieve the project's goals challenging, as well as the deliberations on the predictable technical gaps of the new technology. This was the context in which the Request for Comments (RFC) originated, a series of documents available for all members of the network to read, discuss, and contribute, perhaps, one of the more enduring legacies of ARPANET to the network culture. Not only was the idea of the RFC a result of collaborative meetings, but the name itself reflects the participatory nature of this computer network's documentation, where standards were openly discussed, developed, and adopted through discussion [Hafner and Lyon, 1994, 144]. In some ways, the RFC also drew on the academic tradition of using open publication sharing, but, at the same time, overcame its rigor by proposing a more flexible, informal approach to documentation [Leiner et al., 2009, 28]. Moreover, the RFC workflow can be seen as a positive feedback loop, in

> which ideas and proposals were freely presented and discussed, reaching a particular technical specification when a consensus was achieved through iterative discussion [Leiner et al., 2009, 28]. Predictably, the result of this openness resulted in the growth of the network, which is further evidenced by the ensuing Internet. The dual nature of the ARPANET effort, as not only an instrument of collaboration but a collaborative undertaking in its own right, was asserted by the sharing the information of its own design, albeit among the future network members. Moreover, the improvements in the network capabilities fostered this virtuous circle, for instance, the shift from a paper-based, one author at a time RFC model to a multi-authoring, geographically scattered, on-line supported model facilitated by protocols, such as the FTP or Email, with the added benefit of automatically producing written records [Hafner and Lyon, 1994, 175]. As the Internet pioneer, Vincent Cerf, commented "[h]iding in the history of the RFCs is the history of human institutions for achieving cooperative work" [Network Working Group, 1999, 6], a spirit manifested in the RFC 1 that welcomed cooperation as long as some rules were observed [Hafner and Lyon, 1994, 144], reflecting, in turn, how the Network Working Group was formed. As ARPANET and other networks evolved in what is called the Internet, the process of creating RFCs became more formal, forcing meetings and precise mechanism of discussion (while preserving the friendly cooperative ethos). Nonetheless, the results and new technical standards remained open and accessible to the academia, private companies, and anyone, which is certainly something to highlight, considering the origins of the ARPANET in the Department of Defense at the height of the cold war [Network Working Group, 1999, 12].

To conclude, some interesting findings can be drawn from the interaction of the three attractors in the development of ARPANET, the first computer network. First is the convenience of the notion of the network compared to the system. As noted by Krieger and Belliger, traditional sociology deals with complexity in macro structures where micro-actors operate [Krieger and Belliger, 2014, 59]. This largely systemic hierarchical view falls short of describing the phenomena associated with the general notion of networks and the particular technological realization of computer networks[10]. Second, and observing the emergence of communities such as the Network Working Group,

[10] As an example of this, and recalling the Network Society discussion in the introduction to this chapter, consider that the power and adaptability of the network are mainly indebted to the fact that the IMP was topology agnostic; that is, it did not require knowledge of the network topology to operate [Heart et al., 1970, 553].

one might say that, despite its governmental origins, ARPANET showed, in several aspects, a behavior more akin to the hacker culture. According to Jon Postel, one of the RFC stewards for almost thirty years, "[t]o my knowledge no ARPANET protocol at any level has been stamped as official by ARPA" [Hafner and Lyon, 1994, 204]. This statement points to –not unlike the MAC project and time-sharing systems some years before– the hybrid assemblage in which officially funded projects provided an environment where open collaboration could flourish. Perhaps ARPANET's initial announcement summarized not only the spirit but the possibilities of the project in which the software and network cultures would coalesce through collaboration. As stated by Roberts, "[a] network could foster the 'community' use of computers. Cooperative programming would be stimulated, and in particular fields or disciplines, it will be possible to achieve a 'critical mass' of talent by allowing geographically separated people to work effectively in interaction with a system" [Roberts, 1967, 1].

4.4 Node 3: the multiplicities of Unix

Through the lens of collaboration, the UNIX operating system played an essential role in the network narrative. If it was described as the ultimate embodiment of time-sharing and early collaborative practices in the software culture in the previous chapter, it is not surprising that it was also instrumental in the emergence of the network society, from a technological and cultural perspective. In fact, as Castells pointed out, UNIX was a key factor in the emergence of the Internet [Castells, 1996, 352]. As also observed by one of the best-known historians and de facto proponents of open source and UNIX, Eric S. Raymond, UNIX, and the Internet culture are linked and, in some respects, could be perceived as the same[11] [Raymond, 2003, 27]. Although diffuse, the notion of culture is convenient in locating the interrelationship between software and networks. More precisely, and referring to the concept of technology discussed in Chapter Two, these technical cultures are distilled through praxis, not through planning; they are part of the arrangement I have been calling the socio-technical network. The UNIX culture is

[11]This could appear as a bold statement that needs further clarification. From my perspective, the importance of UNIX in networks is two-fold. First, it is embedded in the network's infrastructure; it could be said that a great number of servers, middle gear, and end-user devices (laptops or mobile devices) run on a flavor or descendant of UNIX. Secondly, the popular discourse about the massification of the Internet in the 1990s about openness and horizontal communications resonated with some of the precepts of the UNIX philosophy. These aspects will be discussed throughout the section.

also part of software culture and, as will be argued, its properties associated with a hands-on imperative and the agency of free-flowing code granted it a symbiotic closeness to the network culture. They not only informed the development of computer networks but also took advantage of it.

However, the relevance of UNIX in the emergence of computer networks (particularly ARPANET) was not straightforward and cannot be attributed to a single motive. Therefore, the evidence presented does not imply the determinist argument that because UNIX was a kind of collaborative endeavor, it was adopted by ARPANET and later by the Internet. On the contrary, and far from a single cause and effect rationale, the development of UNIX provided an assortment of fragmented features that were not tension-free, but whose interaction within the socio-technical network showed certain advantages for networking. Although UNIX and ARPANET shared the same year of official creation (1969) and both featured enthusiastic communities and a hands-on approach, from an engineering point of view, they were essentially very different projects. While ARPANET was a government-backed project with a large budget and high anticipation, UNIX emerged from the ruins of another project, which was nurtured at the beginning without much knowledge of AT&T, who, due to legal restrictions, had no interest in it whatsoever. One might ask, how did this time-sharing operating system (UNIX) evolve in a decade from being a toy project of a group of researchers from a private company into a fundamental apparatus of the Internet? In providing a possible answer, different decisions concerning UNIX reveal several aspects that will influence the outcome of the ensuing free software and open-source software initiatives, shedding light on the issue of software collaboration at the origins of the age of networks. Similarly, the communication imperative is reasserted in the discussion of ARPANET, the Internet, and UNIX when we observe how technologies, such as the computer and computer networks, regardless of their initial pragmatic motivation (calculation on computers and sharing resources on networks), introduced the computer networks as a media machine where different technical cultures met through language.

Because ARPANET's purpose was to share programs and hardware resources among the participating members, it had to connect these parties' different platforms. Therefore, and looking back at the historical moment when networks emerged, it can be explicitly said that it had to accomplish this task by making communication between time-sharing systems possible. Although it could be inferred that, as a time-sharing operating system, UNIX could be a candidate, it must be noted that, at that time, UNIX was still underdeveloped and incapable of performing several functions, let alone talking to other

systems. This is where language comes in as an abstraction whose evolution and stacking gave new software possibilities. Whether as argued in the previous chapter and based on some of the premises of code studies, the software language substrate and its changes resulted in some advances (interactivity and structured programming) in computing, another well-suited benefit for networking, namely, compatibility, came from a decision on language (computer language) by the designers of UNIX. Before UNIX, all operating systems were written in assembler and therefore tied to a particular hardware architecture [Das, 2012, 14]. This decision obeyed two interrelated reasons: a) high-level language compilers produced inefficient code and, therefore, b) assembler was the only choice available to develop an operating system that gave precedence to speed of execution, something that became more evident with the coming of interactive systems. However, around 1973, the team behind the incipient UNIX considered that compiler technology was sufficiently developed [Raymond, 1999] and "[t]o make UNIX portable, they rewrote the entire system in the C language that was invented by [Dennis] Ritchie himself" [Das, 2012, 15].

This language switch was a major step that expanded UNIX into different platforms and, simultaneously, fostered a growing community around it, as described in Chapter Three. The C language itself, developed as a side project by one of the UNIX creators, would also evolve into a standard. As a result of this entangled development, it can be said that "UNIX has probably supported more computing than all time-sharing systems put together" and "it is now hard to imagine doing software engineering without C as a ubiquitous common language of systems programming" [Raymond, 2003, 28]. This affirmation indicates the success of the decision to move away from the constraints of different local assembler language by using an allegedly high-level language[12]. In addition to the C language, the adoption of UNIX is also rooted in its design philosophy. This philosophy favored a modular toolkit approach, where the elements to flexibly build different applications were provided, making the system suitable for developing networks. The UNIX reinforced an "open shop computing model", where the programmer had enough freedom without being constrained by pre-designed limitations. This philosophy of providing the tools but not the final implementations, in short, "mechanism not policies", gave UNIX a competitive advantage [Raymond, 2003, 30,31]. This coincided with Licklider's original concern in the early

[12]I am well aware that by some conventions, the C language does not comply with the definitions of a high-level language, as discussed by Wirth in the section on the software crisis. I will elaborate on its importance in the protocol discussion at the end of the chapter.

'60s when he envisioned a network that encouraged communication between systems without infringing on each system's independence.

To conclude this part, UNIX was a fertile ground for the interoperability required by networking. However, despite its compatibility and philosophy, its widespread adoption was not a seamless process. In fact, at the time when ARPANET ran on the NCP protocol, UNIX had an incomplete implementation that lacked servers of the required Telnet and FTP protocols [Chesson, 1975, 66]. If the seminal 1975 RFC 681, describing the development of the UNIX network unleashed the potential of the system [Salus, 2008, 18], it would take a new protocol (TCP/IP) and a new breed of UNIX (BSD) to operate an additional mix of network and UNIX cultures.

Having said that, the development of the BSD Operating System and the TCP/IP protocol changed the network affordances of UNIX. The year 1983 marked a milestone in the development of computer networks. ARPANET and other packet-switching initiatives of the Internet's formative years switched from the NCP to the TCP/IP protocol to enable communication between the growing number of networks. This change further promoted the adoption of UNIX, specifically, the academic-based BSD (Berkeley Software Distribution) version, following DARPA's (the former ARPA responsible for ARPANET) decision in 1980 to fund the improvement of the system's 3BSD through a contract awarded to the University [Salus, 2008, 44]. It had already become visible to DARPA that ARPANET had become chaos to all parts of the network (mainly universities) requiring a software infrastructure renewal [McKusick, 1999, 22,23] or even hardware update, considering that after a decade in service, the IMP was becoming obsolete [Hauben, 1998]. As a result, and given UNIX's growing acceptance, the Department of Defense (head of DARPA) included the first production of the TCP/IP stack in the already described contract, explicitly selecting the BSD because it was a UNIX, largely *open-source*[13] [Raymond, 2003, 32]. The significance of UNIX in the network narrative is evident here. It not only provided a standard to the heterogeneous set of nodes comprising ARPANET but in doing so, it showed the convergence of the network and software cultures through the UNIX culture, based, in part, on the common ground of collaborative practices. In other words, the developing Internet protocols were in tune with the UNIX philosophy, featuring the availability of code and the interoperabil-

[13]This statement has some nuances. At the time, the concept of open source had not yet been invented, but as noted below, the BSD and its community displayed several features that preceded the concept, compared to the other alternative (AT&T).

ity granted by portability [Salus, 2008, 19,20]. However, as already stated, this transition was not directly implied by the potentials of UNIX. Although "UNIX was chosen as a standard because of its proven portability" [Hauben, 1998], it was adopted because of the lack of another alternative than for its qualities [Hauben, 1998]. Before the implementation of the TCP/IP was finally launched in 1983, with the BSD 4.2 version, "UNIX had had only the weakest networking support" [Raymond, 2003, 60].

Returning to the question expressed in the first paragraph, addressing the BSD as a particular UNIX explains the success of UNIX and its significance in the development of modern open, collaborative patterns in the software culture. First, the academic origin of the BSD must be mentioned. As described in the previous chapters, there is evidence of the collaborative spirit in academic endeavors. This spirit persisted in the Berkeley team's decision to continue researching its version of the system, while AT&T (its original creators) chose to focus on its stable marketing versions [McKusick, 1999, 22]. Second, and on a related note, from the outset, the BSD demonstrated the benefits of code forking in contributing to a more targeted community compared to the struggling AT&T, which was limited to commercialize it. In fact, distant communication was presented from the beginning, with Ken Thompson helping to debug the first UNIX installations in Berkeley from his workplace in New Jersey [Salus, 2008, 41]. Third, and building upon the premise of remote computing in the early days of networking embodied by the ARPANET, the BSD already showed the potential to use networked development. After its origin in Berkeley in 1975, its creator Bill Joy implemented a development model that not only included his team's free-flow of improvements but also encouraged users across the country to submit their feedback, reaching the previously mentioned (and influential) 3 BSD in 1979 [Salus, 2008, 44]. This seed would bear fruits in the following decade (around 1989) by leveraging an already established network culture. The BSD pioneered networked development efforts and networked software distribution via FTP [McKusick, 1999, 25,26]. Given the increasing legal problems with the AT&T code base, following the end of the UNIX commercialization restrictions on the company, and the success of the DARPA-backed TCP/IP implementation, the BSD team, began to build upon this free TCP/IP code base and form, which developed an entire UNIX operating system from scratch without using the original AT&T code. Even though the effort was successful in the end, the continuing disputes with AT&T (now resolved) prevented it

from being more influential[14].

UNIX is a prominent and successful example of collaborative practices in software culture and its associated network culture. Its longevity through the different stages of computing and operating systems history corroborates the practical benefits of the approach, from justifying time-sharing in the late '60s to providing compatibility to emerging networks in the late '70s. In the 1980s, it was the platform of choice of the Internet. In the 1990s, its multiple clones supported and embodied the ideal of distributed development. In the 21st century, it can be said that UNIX is widely used among the most heterogeneous platforms. Perhaps, except in the case of bundled commercial platforms, it has maintained an aura that approaches it to the notion of collaboration in software. As mentioned, with the software toolkit, the UNIX philosophy represented the distributed open-source agency, the logic of "mechanism not policies", and the pragmatism shown by its practitioners and advocates, which gave the system this status as a technology with the potential to foster software collaboration.

From a more ANT-related perspective, the socio-technical network set in motion by UNIX can be described as a collective composed of an active community and a complex arrangement of hardware and software. Here, once again, communication, mainly through computer-mediated language (discussions) or computer language (such as C), worked as an attractor in which the different assemblage registers interacted with one another collaboratively. Consider the communities that emerged to contribute further developments in the operating system. For instance, USENIX, which appeared in the late '70s, sharing insights and new features, given the lack of support by AT&T, or the CSRG (Computer Systems Research Group) at Berkeley, to oversee the innovation needed to meet the DARPA contract requirements [Salus, 2008, 45]. It is also worth noting that although the TCP/IP specifications were open access, given that they were published in an RFC document available on ARPANET, only its implementation on UNIX succeeded, supposedly because of its "robust tradition of pair review", which was not featured on other systems [Raymond, 2003, 32]. It can also be said that this tradition was manifested in early network-mediated communication. For example, in the open discussion conducted on ARPANET or the early USENET on how to plan the switch from NCP to TCP/IP using UNIX, a discussion led by

[14]UNIX's significance was also demonstrated by the development of systems similar to UNIX, like the GNU project and later, Linux. Computing folklore says that had Linus Torvalds been aware a year earlier of a project such as FreeBSD, he would not have started his own project [Moody, 2002, 65].

network mailing lists like "TCP/IP digest" and "UNIX-Wizards" [Hauben, 1998].

However, if communication between human parties using computer networks was fundamental, developments in computer language and processing were no less important. I am referring specifically to the "patch" program, a feature of the UNIX toolkit developed by linguist Larry Wall in 1985. As noted by Raymond, this program perhaps "did more than any other single tool to enable collaborative development over the Internet" [Raymond, 2003, 63]. It allowed outside programmers to merge code improvements into a codebase seamlessly. This program not only represents the UNIX toolkit philosophy of small and efficient programs, but it also highlights UNIX's intertwined nature. It allowed the incremental development and improvement of code between developers (person-to-person) and made possible specific automated operations on distributed code. The network culture, which embraced open collaboration on software, was understood as an affordance of this array of actants. In doing so, the path to modern movements, such as Free Software or Open Source Software was paved, while consolidating the computer and the network as a communication media.

4.5 Node 4: USENET for the people

This section succinctly describes USENET, a computer network that emerged in the late 1970s and played an essential role in consolidating open collaboration practices in the already existing entanglement between the software culture and network culture. Although USENET was another network among the various instances that emerged from ARPANET, the first successful network, it had some particular qualities that were relevant to networks in modern software collaboration. At the same time, it set the dominant narrative of ARPANET's preeminence into perspective. More precisely, there are some key differences between the two networks, and as might be argued, there is an important and sometimes unacknowledged background to USENET that informs the modern rhetoric of network-based collaboration in software development. USENET flourished roughly from the early '80s to the mid-'90s when web-based Internet communication took over. This coincided with other developments, such as developments in hardware like the introduction of the personal computer into the market and network infrastructure like the nascent Internet's switch from NCP to TCP/IP protocols, as well as milestones in the hacker culture, such as the introduction of the notion of free software and achievements in pragmatic systems like as the appearance of

the Linux kernel.

However, before proceeding with the history of USENET and how its tropes relate to open collaboration practices, and its consolidation with the FLOSS movement, it would first be appropriate, to address some fundamental differences between ARPANET and USENET. First, USENET can be considered part of the Unix culture. Even though, in the end, it evolved into a more general communication medium, from a technical and philosophical perspective, it was firmly rooted in the operative system. ARPANET, on the other hand, which was born in the same year as Unix, was not specifically linked to any operating system, and only since its evolution into the Internet was it related to the BSD Unix through its TCP/IP implementation. Second, both networks had a radically different development. While ARPANET was a government initiative, funded by the Department of Defense and its ARPA Agency, the USENET emerged, in part, out of the necessity to connect the large community excluded from ARPANET because they had no projects funded by the DoD (Department of Defense). Thirdly, and combining the previous points, USENET can be considered a grassroots effort, indebted to the voluntary labor of Unix enthusiasts who envisioned a network to share technical information about the system. These enthusiasts were individuals or institutional employees without the political connections or financial resources required to be a node of ARPANET. It is common to find USENET referred to as the "poor man's ARPANET", as famously claimed by one of its pioneers, Stephen Daniels, who clearly expressed this sentiment of exclusion [Daniel et al., 1980]. However, it would be a mistake to portray these networks as opposed because they both emerged partially from Universities [Arns, 2002, 17]. In summary, and as discussed throughout this chapter, software collaboration was closely related to the emergence of networks, which, in turn, displayed a collaborative pattern in their early development processes. However, USENET, more than ARPANET, underscores the grassroots DIY ethos often associated with the beginnings of modern software collaboration, as understood by the free and open-source software movements[15].

The origin of USENET was made possible by a Unix operating system protocol developed in 1976 at AT&T, which was included since its seventh version (v7): the Unix to Unix Copy Protocol, or UUCP [Arns, 2002, 15]. This protocol appeared as a simple way to connect two Unix machines and pass

[15]There is also, of course, a growing relationship between these movements and the normal software industry, a relationship that is sometimes overlooked by activist narratives, but is more related to the notion of open source, hence, it will be addressed in a following section.

files between them [Hauben et al., 1997, 33]. It could be regarded as a manifestation of both the logic of software tools from the Unix philosophy and the communication and cooperation rhetoric of its founding fathers. Possibly aided by the popularity of the system prior to the period of its licensing problems, it acquired a following and a community eager to discuss this operative system that had the potential to connect different combinations of hardware with a Unix version using the protocol. Shortly after the release of the mentioned UNIX 7th version, USENET was born, in 1979, as a network that connected Unix machines, initially to exchange technical information on this specific operative system. In fact, their relationship was so close that it was first presented at one of the USENIX meetings in 1980 [Kehoe, 1993, 30]. The network was created by university students Tom Truscott, Steve Bellovin, and Jim Ellis attempting to connect computers at the Universities of North Carolina and Duke in the United States and provide an alternative to institutions outside ARPANET. As noted, USENET can be seen as an extension of the UNIX philosophy carried out in the realm of a new technology represented by computer networks, and particularly the Bulletin Board Systems (BBS), which was the feature that put USENET on the map. USENET pioneers highlighted their network as a UNIX user's network, a colloquium for "UNIX wizards" where they could obtain and share key information on the operative system, emphasizing, again, the community as its motivation [Pfaffenberger, 2003, 23].

Beyond these principles, and in technical terms, the USENET also reflected its UNIX heritage in other ways. For instance, although USENET first appeared as a combination of basic UNIX tools (the "find" command, the UUCP protocol, and shell scripting), it was later coded –like its host operating system– into the C programming language [Hauben et al., 1997, 24]. This move caused the portability that enabled the presence of USENET in several platforms [Kehoe, 1993, 31]. Also, from the code point of view, the successive rewrites in the "News" software (the layer that allowed the community, not only the computers, to communicate), in its incremental versions known as A, B, and C, required different programmers to complete the work of previous ones. This indicates the agency of distributed code, provided by the free-flow of version A (the first one) of the News software, which was, in turn, released into the public domain[16] [Kehoe, 1993, 30]. Finally, the USENET was not only about software and UNIX. The modem, a piece of hardware, played a

[16]The fact that one of the developers of the News C version (1985) was an employee of the Zoology Department at the University of Toronto –something unthinkable in the ARPANET environment– again illustrates USENET's more democratic nature and the result of the access to USENET code.

fundamental role. Not unlike ARPANET's IMP, this device was needed to provide the links between computers. The first USENET experiments featured homemade auto-dial modems [Hauben et al., 1997, 24], which resonates with the DIY ethos associated with the grass-roots, bottom-up approach presented by a significant portion of the collaborative/open technology rhetoric. As more recent off-the-shelf modems became more widely used, another difference between USENET and ARPANET was revealed, namely, the use of hardware as a commodity (modems and the new personal computer) compared to hardware as limited-access research projects, such as ARPANET's IMPs. Both instances represented socio-technical networks made up of assemblages of communities, software, hardware, infrastructure resources, etc. but, because of historical reasons, they represented the collaborative rationale and the aspiration to the potential of computer networks in a different way.

The evolution of USENET portrays the value of communication (between the socio-technical set) from two sides, the pragmatic perspective of improving a specific technology (as a legacy of UNIX philosophy) [Collyer and Spencer, 1987, 5] and the sociological perspective of creating a community around a common goal. Here, the metaphor of the social operating system offers a label for entanglement, where technology and a social group coalesce. From pragmatism, and its priority of just making a technology *work*, USENET, in its early stages, can be seen as an autonomous autopoietic technology, whose main objective was to improve and diffuse the News software code itself, the neural part of the mainly Unix-based assembly that informed USENET. Not only was a free-flow of code required, but also a protocol for design decisions in which technology and deliberations went hand in hand. In this sense, USENET went a step beyond ARPANET. While the latter relied on mailing lists to fulfill the coordination purpose, USENET relied on the Bulletin Board System. By using them (BBS), USENET introduced interactivity as a feature through which discussions could be held almost in real-time through its News software and the communication block represented by the "post". Clearly, the communal and collaborative spirit embodied in the RFCs, and what their very name represented was carried over by both approaches [Hauben and Hauben, 1998]. The other perspective mentioned, as assumed in the socio-technical association called "the social operating system", was the emergence of a community around the improvement of USENET, which later evolved into a more diverse community, in which technical know-how combined with communication interests, such as popular culture, politics, etc. As with ARPANET, a change of protocol propelled cooperative discussion within the USENET community, seeking to

decide the technical particularities of the switch from the UNIX UUCP protocol to a more news specific NNTP protocol [Pfaffenberger, 2003, 31]. Once the protocol change was made, USENET progressed to a fully deployed communication medium, with discussion groups organized in a well-established hierarchy of several topics. The discussions were conducted through postings and response threads, similar to what would later be known as an Internet forum.

To conclude, although the use of USENET declined in the 1990s in favor of the actual Internet, with the progressive expansion of the WWW, its impact on software collaboration practices is considerable and should be evaluated. Although complex, the influence of USENET can be seen in two ways. On the one hand, at the community level, it directly encouraged the exchange of code and comments in the tradition of UNIX [Flichy, 2007, 53]. It also functioned as a channel for discussion and collaborative development, as Richard Stallman's seminal post or, later, Linus Torvalds' show. On the other, and more on a cultural level, USENET established several trends in Internet communication. They included the fragmented and seemingly chaotic exchange of messages that characterized some software development communication, an issue that Eric S. Raymond attempted to theorize in his book, “Cathedral and Bazaar”, with mixed results; a later section will discuss this topic. Not surprisingly, some of his pioneers deemed USENET “anarchic”; this resonates with some of the political discourses sometimes associated with FLOSS. In short, the combination of community and a networked interactive communication platform can be seen as a precursor to today's collaboration artifacts such as forums, issue trackers, or even version control managers, as they are known today, which have created a new kind of collaboration, “[an] experience of sharing with unseen others, a space of communication” [Wilbur, 1997, 13].

4.6 Node 5: “Commiting” Open Source Software

Open source is, perhaps, one of the most influential terms in the recent history of collaboration in software culture. In the two decades since it was first coined (1998), it has become so pervasive and essential in discourse that it informs most of what has been called “the software metaphor” throughout this work. It is this term and not, for example, “free”, which has extended into various fields as a loosely constructed meta-model of fabrication, politics,

or cultural production, to name a few, that has encouraged networked collaboration, access to context-dependent source code, and a non-traditional conception of intellectual property. With this in mind, several questions emerge. What is the appeal of the open-source construct, and how can it be framed from the network society perspective that is the focus of this chapter? How does it relate to other collaborative paths in software culture? What are the particular characteristics of the open-source development model from a software engineering perspective? This section will address these questions by expanding the historical narrative to include a reflection of the modes of technological production in a networked environment, resonating with the positions stated in previous chapters.

As with other movements and milestones related to the creation and maturation of collaborative practices in software development, one operating system played a key role in the emergence of open source, the Linux Operating System. This observation immediately brings to the fore the tradition and design recommendations embodied in the "UNIX philosophy," as well as its interpretation of software engineering tropes, such as modularity and simplicity. This wording, which is commonly used to describe Linux, points to a *saga* that goes back to the hacker culture, from free software to the classification of open source software (Linux as its most prominent example). It does not, however, imply an already alleged linearity in software history. Specifically, the history of Linux and open source, in general, reveals the fragmented nature of the different paths and approaches to open software collaboration. To support this claim, let us consider two indications of this fragmentation. At the time when the operation of Linux was conceived, the market for the UNIX operating system was atomized and full of incompatible versions. As a result of Linux's success and the introduction of the open-source concept, the *free software* community was divided into two factions, one of them the *open-source* community; that is, two different rationales for one body of common practices.

For the sake of argument, the following oversimplified posit is plausible: of the two instances of fragmentation, surrounding the Linux history, one can be seen as a cause and the other as the result of its history; this requires further discussion. First of all, and after two decades of prosperous and flourishing development, UNIX was facing the problem of too many options with different implementations, none with a decent share of the market. By the early '90s, UNIX was in a difficult place as a result of the previously mentioned decisions made at AT&T and its legal issues with Berkeley and its BSD variant, which still carried the collaborative and hacker ethos. In

this regard, McVoy recommended that the UNIX industry should look to the free software model in order to survive, he stated: “Unix has ceased to be the platform of choice for the development and deployment of innovative technology” [McVoy, 1993]. He also recognized that standards were not enough, as the vendors thought. He defended UNIX, on a pragmatic basis, contending that it was the only platform to achieve interoperability [McVoy, 1993], something that the Internet would later demonstrate.

Consequently and looking at pragmatism, a group of industry and professionals began the search for a new, more business-friendly term that would encompass all the practices, so far, under the “free software” concept. Lead by Eric Raymond, who campaigned against the term free software and the figure of Richard Stallman [Berry, 2008, 122]; the initiative wanted to capitalize on Linux’s success in the software industry and, especially, its distributed production mode, based on access to the source code. Encouraged by the confidence provided by the announcement of the then relevant Internet company Netscape to make the source code of its navigation software open [Moody, 2002, 167], the group devised a set of possible alternatives. Open source was finally selected in the freeware summit in 1998. As mentioned, technically speaking, there is not much difference between what free software or open-source represent; however, as Richard Stallman pointed out, when he referred to the GNU project as “pragmatic idealism”, both terms stress different perspectives. While open-source takes the practical business side, the original free software remains faithful to its principles of free speech and liberty. In the end, it can be said that the open-source movement proved effective, as the term was widely adopted and is more commonly used metaphorically in other fields.

Without overlooking the already addressed discontinuities, it can be stated that Linux was greatly influenced by the UNIX operating system and the free software movement. On the one hand, its creator, computer science graduate, Linus Torvalds, favored UNIX over the other systems. However, in a gesture reminiscent of the old batch/time-sharing dilemma, he opposed the bureaucracy that surrounded the system at his University in the early 1990s [Weber, 2004, 54]. Had it not been for UNIX’s legal problems at the time, Linus would have probably supported a variant of UNIX instead of starting his own project, as he attested regarding the 386BSD system, a BSD descendent in the same league (386 processors) of Linux that was released when he was working on his first versions [Moody, 2002, 65]. On the other, in his original 1991 post on USENET, announcing the operative system –not yet christened Linux– [Torvalds, 1991], Linus referred directly to

the GNU system being developed by the Free Software Foundation. Later, in fact, he would employ the GPL license developed by this foundation in his system, perhaps taking advantage of Richard Stallman's presence during a visit to Finland at the time when Linux was being developed [Berry, 2008, 116]. Herein specifically lies the significance of Linux; it was the first large project that worked (successfully) in a distributed fashion [Salus, 2008, 116]. Although Richard Stallman had taken the same path a few years earlier (also using USENET), his GNU UNIX-like system was stagnant. It lacked a kernel, a component that Linus would develop collaboratively in a network. This achievement signals a more mature network culture, with the Internet, which absorbed USENET, about to make a massive entrance into the public domain, heralding the beginning of the network society. It also pointed to the advantages of software reuse. As history would have it, this young student, Linus, decided to hack a usable operating system "just for fun" –in his own words– and apply Minix (educational UNIX-like system developed by Andrew Tannenbaum) design concepts and porting and integrating tools developed by the GNU Project. His call for collaboration, posted on the *comp.os.minix* USENET newsgroup, benefited from a community eager to contribute to projects other than the legally contested field of UNIX [Weber, 2004, 50]. It also demonstrated the Internet's relevance as a communication system for setting up socio-technical networks. For instance, the Linux kernel code had these features; A, it followed a standard obtained electronically (POSIX); B, it was published from the first version on a free downloadable FTP site; and C, it received bug fixes by E-mail [Chopra and Dexter, 2008, 15].

This process can be considered the first phase of the Linux Operating System. It emerged from an individual effort that evolved into an iterative and collaborative cycle using the Internet and based mainly on volunteer work, and, progressively, it would develop all the parts of an operating system (graphics, networking, utilities, etc.). The second phase marked the consolidation of a model embodied by the open-source concept and its embracing of pragmatism. At this stage, and still headed by Linus Torvalds, Linux surpassed its humble student origins to become an industry-adopted system, developed by programmers and paid by major software companies [Bauwens, 2014, 104]. The newly-born open source community strongly criticized the free software movement based on the drawbacks of using the adjective "free" in the business world and the movement's associated fixation on political issues, rather than to stress the alleged technical superiority of a distributed collaborative effort, arguably the main focus of the whole idea of open source. For instance, although Linux shared with GNU the practice of distributing source

code , it was more a practical strategy to exploit the potential of talented programmers worldwide [Chopra and Dexter, 2008, 15].

The departure of open source from the free software corpus was materialized through two instances, the Open Source Definition (OSD) and the Open Source Initiative (OSI). The open source definition basically stresses the same points on source code of the famous four liberties of free software (free distribution and allowing derivative works), adding a pair of anti-discrimination clauses (a smart move considering the globality of the Internet) that can also be considered akin to the ethos of free software. However, it departs from free software because it is not bound to a particular license and chooses pragmatism over idealism [Chopra and Dexter, 2008, 16]; on the latter, it directly declares that “license must be Technology-Neutral” [Open Source Initiative, 2007]. Meanwhile, the open source initiative grouped all the actors interested in achieving a more professional profile for this software; it was instrumental in its promotion based on its mode of production, removed from any political discourse. In brief, “OSI [open source initiative] intends to supply merely a software engineering methodology” [Chopra and Dexter, 2008, 35]. Ultimately, it can be argued that the purpose of promoting Open Source in a broader community (particularly business-oriented) was accomplished; the result was the adoption of a new term, coined to describe a new form of production based entirely on the possibilities provided by the Internet. As stated in the mentioned Netscape announcement disclosing its source code, a commercial move similar to McVoy’s recommendations on “sourceware” which in part triggered the Open Source movement, the idea now was to harness “the creative power of thousands of programmers on the Internet by incorporating their best enhancements into future versions of Netscape’s software” [Netscape Communications Corporation, 1998].

To conclude, a brief assessment should be made on the relevance of open source in the articulation of an already established set of collaborative practices in software development and the new capabilities of networks (particularly USENET and, later, the Internet), understood as a mature technology that extended beyond the academic community. In this regard, open source’s entry into the discursive universe of software history and specifically of collaboration brought back critical assumptions that informed the software field. Issues concerning craftsmanship, design versus a hands-on approach, and software engineering were reproduced considering the influence of networks. Linus Torvalds, a student working alone outside the usual (then) software production circles, became the romantic figure of the programmer-artisan. This time, however, the entrance of the networks allowed not only one per-

son but a group of them, interconnected by the Internet, to operate in a distributed way. Thus, the craftsman/artisan perspective, although problematic, can be reinterpreted within the framework of the collective already discussed, an intertwining of community and technical infrastructure in which the imperative of technology, not only as a tool but as "a manifestation of human creativity and expression" [Weber, 2004, 47] can be fulfilled. With this in mind, Linux can be appraised as a piece of software or a "distributed code object" [Mackenzie, 2006, 69] produced by distributed craftsmanship, where the agency of the participants was shown thanks to the communication layers of the Internet. On a related note, this distributed craftsmanship does not entail technical virtuosity per se. In the case of Linux, once again, its kernel design reflects the dispute between hackers and planners, a key component of the hacker imaginary of technical folklore, on one side, and the struggles within the genesis of the software engineering discipline on the other. Accordingly, Linus Torvalds was an exponent of the hands-on approach to software development, which can be seen as an example of the general "make things work" engineering philosophy advocated by FLOSS [Berry, 2008, 102]. More accurately, the Linux kernel presented a monolithic design, a choice based on pragmatic reasoning that caused plenty of acrimony on-line. Notably, for academic and Minix creator, Andrew Tannenbaum, it represented a leap backward compared to modern microkernel design [Berry, 2008, 118][Weber, 2004, 99].

Similarly, and reflecting the same working ethos, but in network design, the implementation of the TCP/IP in Linux was achieved through scattered iterations without a thorough design from the beginning. The previous manifests the tension between the concepts of "make it work first, then make it work better" and the "throw away the old and write all from the bottom up for a perfect grand vision" with Linux and the open-source model primarily favoring the former [Weber, 2004, 104]. Ultimately, these lines of reasoning, which frame open source from the network perspective, coalesce in the movement's reading of software engineering. From the beginning, open-source claimed the alleged superiority of its model [Chopra and Dexter, 2008, 21], which, as mentioned, is merely a methodology composed of quick and frequent code releases in which users are involved early in the process [Chopra and Dexter, 2008, 35]. Linux is, therefore, important to the open-source movement because it clearly presents an unorganized development that was distributed geographically using the Internet [Salus, 2008, 116]. Perhaps this was most accurately described by Eric S. Raymond's colorful statement in his famous "Linus law" that "given enough eyeballs, all bugs are shallow"; this is also described in his renown "The Cathedral and the Bazaar" [Raymond, 2001,

19]. This description pointed to the power of an interconnected community focused on a common goal and a common set of shared source code, a symbiosis that I have called before "the distributed agency of source code". It was a reverence to efficiency above politics and idealism [Berry, 2008, 122] and the ultimate systematization of collaborative practices in software formalized in the ways described in the following chapter.

4.7 Node 6: Debugging the bazaar

Following the last section, one of the manifestations of the open-source movement understood as a culture was the defining concept of "bazaar". The term was coined by Eric S. Raymond, a key actor involved in defining open source, in his 1999 seminal and highly criticized text entitled "The Cathedral and the Bazaar" [Raymond, 2001]. This work can be seen as part of a defining metaphor that appears in opposition to a counterpart complement, represented by the figure of the "Cathedral". The image of "Bazaar" is proposed to describe the modes of software production, namely, open-source software, operating horizontally in the technical infrastructure provided by computer networks (mostly the Internet). Meanwhile, "Cathedral" represents hierarchical software production where there is a definite work division, as is often the case (at least in Raymond's argument) with proprietary software. The tension embodied by the Cathedral and Bazaar duple is important mainly for two reasons. First, from a historical stance, the essay was an instrument for a community to define itself and gain recognition and, second, from a technical perspective, the essay established several controversial points and arguments presented as evidence of the superiority of open-source software as a product and a process. However, the idea of the Cathedral and Bazaar, in the last point, may prove irrelevant in an academic undertaking, considering the journalistic tone and lack of evidence in Raymond's text. Nonetheless, and despite the informed criticism through the years, the idea of the Cathedral and the Bazaar continues to be used to frame several topics of open collaboration involving software engineering applied to this kind of software. Furthermore, the various shortcomings of Raymond's position reveal the complexity of FLOSS development as the already defined socio-technical network that enables the discussion of several tropes of the network culture.

Raymond's extended essay can be considered a milestone in establishing an open-source software culture. It also addresses a number of technical and social issues that, since the emergence of a network society, must be considered in describing open collaboration practices, particularly in FLOSS. To

this end, a brief assessment is made of the "Cathedral and Bazaar" concepts to draw relationships with some points made throughout this work. Overall, Raymond's text involves a somewhat anecdotal analysis of the Linux Operating System, perhaps the most notable piece of open-source software, using personal experiences on his developing of a tiny piece of software called "fetchmail", following his own distilled *open practices*. In the process, he touches upon some points that resonate with the division made in this research between a systemic view of software (as a socio-technical system focusing on modularization) and a networked view of software (as a socio-technical network focusing on communication). In the first case, "Cathedral and Bazaar" point to UNIX's tradition of small tools, rapid prototyping, and evolutionary programming as a decades-long precedent for the practices developed by Linux [Raymond, 2001, 20]. This rationale of using tools to make tools is evident in the choices made by the GNU project to emulate the UNIX system, given its convenience, or by Linus Torvalds' use of the academically intended UNIX clone, the Minix, as a scaffold to produce Linux [Raymond, 2001, 24]. This can be seen as a manifestation of the pragmatic impulse behind the celebrated hands-on imperative, which favors the programming praxis and trial and error approach over detailed planning.

As further evidence, consider the two well-known observations that good programs emerge from personal need (scratching one's back) [Raymond, 2001, 23] and that good programmers know what to write, but great ones know what to rewrite or reuse [Raymond, 2001, 24]. The latter underlies, from my perspective, the systemic view of software that depends on modularity and free interchangeability of artifacts (mainly code), which indicates the modular agency of code (a clear division between code and data structures) [Raymond, 1999, 37] in preparation for the entrance of networks. From the network perspective, the entire essay unfolds as an extended tribute to the benefits of the Internet in developing open-source software, with Linus Torvalds at the center as the revolutionary instigator of a new network mentality in software creation. Here, the modularity inherited from the UNIX philosophy would be enhanced by the new communication possibilities of computer networks. For example, Linux's rapid-release method, which embodies the new possibilities of the Internet as a medium [Raymond, 2001, 30]. Most importantly, however, the Internet not only offered a physical infrastructure for rapid release and access to source code, but it also implied new group communication patterns. Now, new forms of technical and social organization were required, given the possibility of geographically distributed asynchronous software development.

Perhaps this essay's proposed division of the system or the network metaphor, whose organization served the purpose of this work, can be best appreciated by considering Raymond's most famous statement, "[g]iven enough eyeballs, all bugs are shallow" [Raymond, 2001, 20]. This quote reveals Raymond's overconfidence of a new and revolutionary development model that relies heavily on peer-to-peer communication using the Internet. It is one of the highlights of the essay; it suggests that bug tracking and correction operates differently than traditional software development, where maintenance depends solely on the company in charge of the software. Open source development implies a participatory model, in which, as suggested by the text, users should be considered developers [Raymond, 2001, 27], obtaining the valuable feedback that source code access generates. In this regard, Raymond foresaw the possible counterargument of communication overload within large development teams, based on the well-known "Brook's Law", included in the cornerstone text of software engineering, "The Mythical Man-Month" [Brooks, 1995]. Raymond sustained that this law does not apply to the case of open-source software. In other words, the feared exponential growth of communication exchange between programmers means that every peer would have to have contact with each other, something that would be unthinkable on the scale of the Internet. In this regard, Raymond observes that the modularization of software and the organization of core teams considerably reduces communication by parceling the development efforts [Raymond, 2001, 35]. To conclude, Raymond addresses other controversial issues, such as the ego-free nature of programmers and the anarchic reading of the open-source software movement. Most notably, however, is that the whole Cathedral and Bazaar text and metaphor was and is understood by the community as a defense of the technical superiority of open-source software above all other alternatives that do not encourage the open and distributed model based on the Internet.

Although the success of the Cathedral and Bazaar concept has been evident even two decades after its publication, and it is still used as an argument in favor of open source, the text has also been criticized by practitioners in different disciplines. By addressing some of these contestations, the concept of Cathedral and Bazaar can be put into perspective, and the romantic and misleading discourse around open-source software can be overcome. Thus, obtaining a more accurate understanding of the particularities of open and free source software as a product and a process. To this end, and to offer a general view, Bezroukov defines Raymond's main points [Bezroukov, 1999], as the first step to problematize not only the claims but also their interpretation by the community:

- A. Brook's Law does not apply to Internet-based distributed development.
- B. "Given enough eyeballs, all bugs are shallow".
- C. Linux belongs to the Bazaar development model.
- D. Open-source software development automatically yields the best results.
- E. The Bazaar model is new and revolutionary.

This list frames Raymond's thought and outlines the main points that a critique should cover. Of course, this exercise cannot be exhaustive without incurring into the distortion that activism often implies; therefore, here, I address some general, common narratives of the open-source software community with this warning in mind. The objections are addressed following the order above. First, the rebuttal of the application of Brook's Law in open source software is, perhaps, the concept's boldest statement from the perspective of the software engineering tradition. Raymond's concept of Cathedral and Bazaar has been primarily criticized because of its lack of evidence outside of the author's account [Glass, 2002, 175,176], let alone the non-systematic journalistic tone of the writing [Hernández et al., 2008, 105]. The assumption that Bazaar-style development reduces the communication exchanges compared to regular models is misleading, considering that organization in core groups is not exclusive to open source software teams. In fact, despite it being one of the main premises of the essay, all major open-source software are hierarchically structured [Bezroukov, 1999]. This means that software development, in general, tends to fragment communication into groups, using modularization and the mentioned hierarchy.

As observed by Bezroukov, the Cathedral and Bazaar concept provides a "set of surprisingly universal guidelines involving flexibility, modularity, and an aggressive feedback loop that all software developers should seriously consider adopting as their guiding principles" [Bezroukov, 1999]. This establishes the broad applicability of Raymond's arguments. Secondly, the statement that all bugs are shallow is probably the most contested part of the whole argument. Glass openly denounced the fallacy of the argument based on the fact that the depth or superficiality of bugs does not correlate with the number of coders looking for them. Research suggests that an optimal number of code inspectors exists, and exceeding this number does not increase efficiency [Glass, 2002, 175,176]. Several authors have also pointed out the code-centric

nature of Raymond's position. For instance, when using low-level languages or in re-engineering tasks, "the value of higher level documentation, like specification of interfaces and descriptions of architecture, can be greater than the source code itself" [Bezroukov, 1999]; this establishes software as a multi-artifact informed process. Moreover, bug detection alone is not enough, no matter the intensity or speed of the process, when maintenance tools are misused in all development models [Glass, 2002, 81]. Regarding the point that refers to Linux as an instance of the so-called Bazaar concept, its shortcomings are related to the already described deficiencies. The whole argument of Cathedral and Bazaar concept can be seen as an inference move based on two intertwined stages, the personal case of the "Fetchmail" software (developed by Raymond) and the Linux Operating System, which distills the software engineering superstructure embodied by the Bazaar model. One of the problems of this approach is that only two cases fulfill the universal claims of the Bazaar construct, made more profound by the lack of empirical data. In the same line, there is a notable asymmetry of complexity between a one-man effort, like Fetchmail, and a whole operating system such as Linux. Concerning the latter, it is contradictory how it is positioned as the primal example of the Bazaar model, while simultaneously praising Linus Torvalds' leadership and delegation to his so-called "lieutenants".

Relating to the last affirmation, some authors have described Linux as a "benevolent dictatorship", indicating the inconsistency of the Cathedral-Bazaar concept presented as a duality. The last two points (D and E) can be presented together as they refer to a similar issue, the superiority of open source software (the implication is subtle but clear) and the superiority of the Bazaar model. The first statement is addressed by studies that show that the fail rate of open source software is high, and obeys, as already pointed out, to artifacts other than code, such as poor documentation or maintenance [Coelho and Valente, 2017]. In his influential study on the open-source development model, Fogel explains that open source projects fail for the same reasons as others. However, they have the potential of having new problems produced by the idiosyncrasies of the model, which can be dangerous when not acknowledged properly [Fogel, 2017, 1]. The second statement is more difficult to address. It is clear that the Internet and its communication possibilities have generated a complete revolution on how software efforts are conducted. However, these benefits are not exclusive to open source software, although perhaps more noticeable, and while the agency of code sharing and its accessibility is important, it does not meet the comprehensive vision that a development model should offer. Once again, the importance of the Cathedral and Bazaar concept, even with its problems, is more histor-

ical and pragmatic in providing a common narrative on which to build. The characteristics of this important milestone can be seen from a more general perspective where the culture of network and the abstraction of the bazaar model can be extended to evaluate other fields.

As repeatedly stated, despite its inconsistencies, the importance of the Cathedral and Bazaar concept is undeniable. Not in vain, it has been part of the open- source culture for almost twenty years. However, if we intend to analyze the adaptability of the open-source tropes in other contexts, the influential essay must be analyzed. This exercise is useful for three reasons (among others): 1. It allows more accurate characterizations of the relevance of source code as an artifact in software development, as well as its preponderance in the open-source approach to software engineering; 2. It pinpoints the shortcomings and tensions of open source, shedding critical light on its narratives as a development model and a cultural movement; and 3. It highlights the importance of technology-mediated communication in modern undertakings, particularly in software development. This reading has two intertwined interpretations when combined with other claims made in this work. On one hand, from a pragmatical point of view, the bazaar model and open-source software development practices, in general, seem to be compatible with ideas sustained in previous chapters, such as the identification of craftsmanship present in technological endeavors (including software) and the so-called hands-on work ethic, favoring a trial and error approach over detailed planning. The well-defined manifestation of this school of software development is the positioning of source code at the center of development efforts, relying heavily on the availability of the artifact through computer networks and the labor of potential users/coders. On the other, and from a more abstract and ontological perspective, the ideas of Eric Raymond defended in "The Cathedral and the Bazaar", reveal several veins of technological determinism, which are evident in the mentioned overconfidence in one artifact (source code) and its distributed agency as the main requirement that automatically implies a superior software development model. Of course, the open flow source code is necessary, but it is a limited condition, something patent when considering the described Bazaar model's troubles from a theoretical and, more importantly, practical perspective. A valuable conclusion is that the bazaar model is not only a sign of the logics of the network society and of computer networks as a socio-technical culture but that software development efforts, open or not, require the operation of a complex assemblage, namely, a socio-technical network, composed of linked nodes, or actants, exchanging information constantly.

4.8 Establishing a connection: platform network society

The historical approach that has been presented is not intended to draw a causal line of reasoning over time, as argued, this would go against the evidence and methodological framework of this work. Instead, it offers several scattered examples in the evolution of computers, software, and networks that reveal the complexity of this technological development and how some previously addressed issues reappear under a different light provided by the framework of the network society. This last descriptor, namely, network society, entails the great influence that this technology has on the modern world and that, as in the case of software, points to the existence of a culture that surrounds technical artifacts as a socio-technical assemblage. This does not mean that networks have undergone a similar process to software from the perspective of open collaboration and, therefore, can be addressed through a kind of isomorphism. On the contrary, although software was a previous development, networks and software have an entangled history in which tropes, such as a hands-on imperative, materiality, and craftsmanship, are recreated under a common umbrella. Although a certain parallel can be drawn from their common history, in the end, this is done more for the sake of clarity.

To illustrate this mindset, consider the tension between detailed modeling and planning in the development of technology and a more pragmatic approach where advancements are conceived with a practical focus on doing and then organizing. As we saw in Chapter Three, the 1970s brought the C computer language, which over this period, became a widely used language, challenging appropriate layer architecture and allowing low-level access. The same decade also marked the emergence of packet-switched computer networks, ARPANET being the best known, which led to a conflict that I called, using this interwoven history of software, the computer networks crisis.

The computer networks crisis refers directly to the problem that emerged in the second half of the 1970s when the need to interconnect different networks became evident. The solution to this predicament led to the establishment of the protocols of the Internet (the prefix "Inter" prompting the heterogeneity to be resolved) as we know it today. To relate this to the narrative of this chapter, it can be stated that the sections presented are organized around the turning point created by the introduction of a proper Internet. Therefore, examples like ARPANET, UNIX, and USENET can be seen as different manifestations of the open collaboration tropes in isolated

networks before the Internet. Once the Internet-based network culture flourished, more complex socio-technical arrangements were possible. The first examples, for instance, were surrounded by more focused communities, such as government founded scientific endeavors (ARPANET), a privately owned operating system but with an open logic (UNIX), and grass-roots DIY network initiatives (USENET)[17].

On the other hand, open-source and the Cathedral and the Bazaar metaphor point to problems of distributed, loose, and highly heterogeneous communities and their struggles to produce a technical device. Consequently, and as mentioned previously, if the C language in software implementation represented a response to the layered stack structure in favor of a murky trial and error approach, the TCP/IP protocol emerged in a similar way within the framework of the computer network crisis. Simply put, two main approaches provided interoperability between computer networks that could not communicate with each other, the OSI model and the TCP/IP protocol. The first was a multi-backed effort that proposed a well-defined stack of seven layers with distinct network interfaces. The OSI model, however, never took off; it remains an example of an abstract technical construct that failed under its own weight, and it is only used as a reference in academia and industry, but not in practice [Russell, 2013]. Conversely, the TCP/IP protocol, which was originally the TCP protocol (IP would be added later), appeared out of a brief specification and a period of experimentation [Leiner et al., 2009, 25,26]. It was developed iteratively out of the need to substitute ARPANET's NCP protocol, which was not suited or designed to exchange information between networks with different architectures. The TCP/IP, as hinted previously, was an example of a successful technical implementation that did not follow a vertical, over-designed structure. Its mode of production brought forth a more hacker-like hands-on approach. From this perspective, the TCP/IP protocol averted the network crisis by facilitating network cooperation in a way that was not only open but encouraged further open developments. In this sense, even the open architecture of ARPANET's IMPs anticipated the growth of the network [Heart et al., 1970]. This spirit was brought about by the freedom of choosing a particular computer network architecture, given the existence of a meta-level of communication. This is the underlying technical idea carried out by the Internet: an open architecture [Leiner et al., 2009, 24].

[17]The prominent role of the UNIX Operating System - a piece of software - in the three examples strengthens the claim of the close relationship between software and computer networks.

The development of a meta-level in which computer networks with different configurations could interact can be seen as a contradiction with the claim that a layered stack is not a practical alternative to obtain this characteristic. However, this oversimplification would again imply a contested dichotomy in the light of the selected view of technology, or particularly from the ANT perspective. The problem was not between the *hackers and planners*, that is, between the practical trial and error approach and the detailed design, but rather in the logical observation that as a socio-technical assembly grows in complexity, it becomes more difficult to obtain technical (or social) agreement. The success of TCP/IP over the OSI model does not imply the superiority of one approach over the other; instead, it indicates the difficulties in technical negotiations and how they challenge technical determinism. As in the critique of the Cathedral and the Bazaar, the advantages of rapid deployment and low communication load of one approach face documentation and maintenance issues, in some cases, this can be said of networks. The bottom line is that the integration of various computer networks in the second half of the '70s into what would be known as the Internet has been argued as the shift from a systems-to-network paradigm, in other words, to a socio-technical network. The specific meaning of this statement, made throughout this chapter, points to control and communication. According to Cerf and Kahn, in the original proposal of the TCP protocol in 1974, the main target was the flexibility to enable communication between different networks [Cerf and Kahn, 1974].

This led to the enforcement of network autonomy based on four recommendations, namely, 1. each network should have its own configuration, and no changes would be needed on the Internet 2. if a packet does not reach its destination, it should be retransmitted, 3. black boxes should serve as gateways between networks 4. There should be no global control at the operations level [Leiner et al., 2009, 24]. A concern for network independence is evident; however, the last point acknowledges the impossibility of control, one of the fundamental cybernetic concepts, key to understanding systems at the networks level. This is not just the arbitrary operation of changing terms. Although sometimes a system can be described as a network and vice-versa, the full meaning of this shift implies an entirely different paradigm more suited to modern open collaboration in software and other digital dependent fields. If control and human-machine relationships were fundamental in understanding the leap of the computer from calculating to media machine through interactive computing, leading to the change of linear batch to fragmented interaction, the emergence of networks intensified the communication substrate of interactivity. This demanded not only isolated human-machine

communication but additional machine-machine and human-human communication at different times and from distant locations. As a result, technical advances, such as code modularization and software engineering, which aided communication within a socio-technical system, were achieved by an intensification in the communications rationale. This led to new socio-technical networks, where topology and protocols were better descriptors than feedback and control.

To conclude, the historical narrative presented and the construct of the socio-technical networks represent a new agency, following an algorithmic or modular agency, namely, the distributed agency in which artifacts are not subject to total control and are affected by technical and social negotiations within a geographically dispersed infrastructure. This scenario enabled collaborative phenomena like distributed code objects (Linux) or new industry approaches (FLOSS) that consider communications (and protocols) as one of the key aspects of technical cooperation. Therefore, the following section will present the artifacts that make up modern software collaboration, highlighting protocols as communication devices. It will also address ANT's shortcomings, reasserting that communications are more than language operations and represent negotiations between actants in order to fulfill an objective [Farías, 2014].

5 — The nodes of collaborative software development

5.1 Introduction

So far, it should be clear that software, even more so open software, is a highly complex technological development that recreates the intimacy between human and technical devices. It highlights the importance of language operations in artifacts, which, like software, lie between the tensions of pragmatic expectations and symbolic abstractions. Despite the common assumption that software is a particular kind of language, its history and complexity show that this oversimplification, in which code is positioned as the sole artifact, is constantly disputed by different agencies, social forces, and modes of production. It is, thus, better understood as a socio-technical development in which different kinds of text artifacts, besides code, are used. Hence, software is a complex artifact, which involves many artifacts and their reciprocal interactions in time. The evolution of software, presented in previous chapters, illustrates this growing complexity by indicating how the entire assemblage of the computer program could be understood as a system (structure) or as a network (topology). Although selecting this particular point of view could be considered arbitrary, the network perspective has been favored to assess the socio-technical phenomena surrounding software collaboration. The reasoning is that even though the system also reflects complexity, it tends to do so in a more hierarchical and closed way, where control and function are the main interests. Networks, however, allow for more horizontality and openness that require communication between the socio-technical assemblage to achieve a specific objective[1]. Of course, this points to computer-mediated communication (computer networks) and a series of artifacts, which have

[1]It should be noted that modern systems theory envisages different theoretical constructs to address networks. However, the mental image suggested by the concept of topology is closer to networks and, therefore, this term is preferred.

been increased through collaboration in these networks.

Following the history of software, there have been technical developments associated with controlling this complexity. Among them, modules and structure programming, on the one hand (software as a system), and computing resources sharing and geographically distributed and asynchronous collaboration, on the other (software as a network). The latter has been fundamental in developing modern software, particularly open source or free software. The importance of networks also operates at a more general meta-level that provides a metaphor for inscribing several old and new social behaviors, technical artifacts, and communicative negotiations, conveying the broad scope of the construction called network society. In that vein, this chapter addresses modern and open collaboration in software development. It uses two assumptions. The first is that if software is a complex artifact, then, open modes of software production operate as a socio-technical network in which each piece can be seen as a hybrid node-artifact. The second is that each node-artifact contributes a specific task to a unified and topological agency. That is, that a coordinated and iterative but chaotic effort where the node-artifacts depend more on their ability to negotiate certain protocols than on a particular expectation of the outcome of a tandem input/output operation. Consequently, the historical narrative will be set aside to focus on this interrelation element. As anticipated in Chapter Four, models dedicated generally to networks such as ANT can be complemented with the prominent role of communication, and the socio-technical assemblages take the form of the hybrid node-artifacts, where actants of different levels converge. Of course, modern collaboration in software development resorts to a large group of these artifacts, requiring their further selection and organization.

Some clarifications and definitions are required before stating which node-artifacts will be considered. One might think that issues concerning determinism might arise, given the high status of the network narrative, as is usually the case with technical-based metaphors. The same can be said about ideas as the universe being a computer or the assumptions concerning an isomorphism between software/hardware and thinking/brain, which point to valid concerns about oversimplification and dehumanization. However, the network and particularly socio-technical networks can be seen as a complex whole with a significant amount of indetermination provided by its open topology, a strong social component, and even the menaces from different fronts (for example, neutrality on the Internet). Ultimately, computer networks, as a metaphor, are better suited for describing socio-technical phenomena. This statement leads us to the matter of agency and how this

operates within such networks. As presented in previous chapters, the intertwined evolution of computers, software, and networks presents three levels of agency inter-operating in open collaboration in software development. First, as presented in Chapter Two, there is the agency of algorithm, which goes back to the origin of software and the computer as a calculation machine. Second, as presented in Chapter Three, there is a modular agency, which attests to the increasing complexity of software (due in part to interactive computing) and the birth of software engineering. The last one, presented in Chapter Four, is the distributed agency, only conceivable with the possibilities of computer networks under the umbrella of the network society.

These three levels of agency exhibit increasing human participation, the algorithm agency allowing less room for hybrid interaction than the distributed agency. However, they all reveal the tension between programmable media and social idiosyncrasies. These node-artifacts constantly shift between these modes of agency as a result of the actions of the collective, namely, a combination between a community (the labor force) and the limited decisions of technical devices. The coordination of these actions occurs following some protocols, which are also subject to discussion and sometimes not properly arranged, that converge into a common goal, that is, a piece of software. Therefore, software's unstable nature is also transferred into the operation of this particular socio-technical network; this defies linearity and a clear distinction between product/process, as classical software engineering presents, in the open collaboration construct. It, thus, favors a more chaotic yet effective socio-technical network with its nodes-artifacts and their respective modes of agency. The communication protocols help organize and coordinate the unstable topology of the socio-technical network, producing what was called the social operating system. Last but not least, although code and code objects still hold an important place, the architecture of the socio-technical network demands a more integral appreciation of software, thus increasing the power of a single artifact.

Following this train of thought, this chapter is organized as follows. The first section addresses general collaboration in society, setting aside the emphasis on software. Although this undertaking is clearly too broad to be achieved completely, the main objective is to discuss some paths in general human collaboration, based on two assumptions. The first is that collaboration has always been a part of human societies; the second, that collaboration is a hybrid field in which action, agency, and communication play a significant role. This section will provide an abstract outlook, where the hybrid node-artifacts of the socio-technical network used in modern open software

production interact. Provided this abstract context, the following sections will cover different artifacts that are part of the mentioned network. The second section will address the idea of code objects as the functional building blocks of software functionality. The third section will extend the understanding of text artifacts beyond code by discussing community documents, mainly text communications through mailing lists, wikis, and IRC, among others, to provide a comprehensive appraisal of software. Community exchanges will be depicted in the fourth section, focusing on the politics of team organization and leadership in software development. The last section will shed light on the abstract artifacts and how they are embedded in the protocols, standards, and methodologies that emerge as a metalevel of abstractions based on practices of the FLOSS community.

It should be noted that this division, as is often the case, is arbitrary. It attempts to break the complexity of the software development assemblage by selecting specific points of interest. Although other divisions could be set, this one, in particular, meets the software portrait presented in the previous chapters.

5.2 A general approach to collaboration

This section aims to approach the concept of collaboration from an abstract perspective. In doing so, concerns about software are momentarily set aside, and instead, the focus is directed to the structural, historical, and sociological traits of collaboration. Nevertheless, the interest in software will prevail across the entire discussion on collaboration, ultimately, remaining at the core. A general framework is provided to understand the artifacts involved in FLOSS development better and how they interact in a collaborative sociotechnical network. Clearly, assessing a complex issue like collaboration would be impossible in such a limited space. However, the challenge should be reduced by selecting a few key points among those intertwined with collaboration. Thus, three related issues were selected, the importance and structure of collaboration, some historical considerations, and its theorization (or lack of it) in FLOSS. Firstly, we have to acknowledge that theorizing collaboration is difficult [Lovink and Scholz, 2007, 24] and that, although it has been everpresent in history, it is a changing concept. In the words of Geert Lovink, "[l]ife kicks off with collaboration" [Lovink and Scholz, 2007, 9]. Secondly, and somewhat contradicting the last statement, lately, there has been a hype around collaboration seen as a commodity, either as a new experience or as a recovered ritual [Lovink and Scholz, 2007, 10]. Thirdly, this new prominence

can be considered the product of the so-called new technologies. Particularly, how society has perceived free and open-source software. This new framework for collaborative software is based on two assumptions: computer networks have enhanced collaboration, and voluntary work is the main form of work in FLOSS. While the former is reasonable, given the modes of software production, the latter has been proven to be misleading, as it contests one of the founding myths of FLOSS due to market forces. However, this observation allows us to build a more accurate representation of collaboration in the software field.

First off, and from an intuitive point of view, there has been a tension around collaboration when considering economy and history, and therefore politics. Oversimplifying the issue, some interpretations of capitalism have led to a dichotomy where collaboration is opposed to competition, being the latter closely related to the rhetoric of the capitalistic mode of production. However, collaboration is in no way incompatible with capitalism; in fact, it is at the core of many business undertakings. Having acknowledged the inconvenience of these dichotomies, the origin and evolution of collaboration within different modes of production and with culture in general highlights the issue of collaboration as an inherent feature of the human condition. In that sense, Loving's quote from the last paragraph already establishes a close relationship between collaboration and life itself, a premise that is also shared by this work. The previous does not directly imply a natural condition of collaboration –biologically speaking– but rather points to its social condition. Thus, "life" refers to the social nature of humankind, namely, to how collaboration operates within the socio-technical network embedded in every society.

Following this last clarification, the focus will be upon the organization required to collaborate and the artifacts used. Axelrod attempts a *cooperation theory* in which the author points out, among other things, the state of affairs of natural cooperation. The author reminds us of how other authors, such as Hobbes, expressed the implausibility of cooperation among individuals without a central government [Axelrod, 2006, 4]. This observation not only entails a Eurocentric stance, which is understandable given the temporal circumstances but also shows a genuine concern for organization, even if it may rule out other possibilities of non-Western government cooperation, such as cooperation in other cultures. In Spehr's exhaustive study on "free cooperation" –a title that resonates with this work– he mentions political tropes, such as Rousseau's social contract, as a catalyst for cooperation [Spehr, 2003, 11]. Regardless of how he theorizes government, Spehr restates the importance of organization, which is, in fact, political, and the conditions required for a

cooperative praxis between free individuals who negotiate to achieve a goal. He states *"[s]chließlich umfasst eine Politik der freien Kooperation auch eine Politik der Organisierung. Organisierung bedeutet, sich mit Gleichgesinnten (oder besser gesagt: in bestimmten Punkten ähnlich Gesinnten) gemeinsam für bestimmte Ziele einzusetzen und dabei gleichzeitig bereits eine alternative Praxis zu entfalten"* [Spehr, 2003, 52].

Although we now have more elements to explain collaboration, namely, the relevance of organization and its historicity, two aspects are challenging when trying to adapt this rhetoric to the field of software production modes, first, the term itself. So far, most references mention "cooperation" or "collaboration" interchangeably, but some clarity is needed. To this end, it may be helpful to consider Lovink's classification, in which collaboration, cooperation, and consultation (in that specific order) are compared based on priority. This focus favors a spectrum in which collaboration is preferred to consultation, the latter representing a collective environment with communicated peers and infrastructure but without coordinated efforts [Lovink and Scholz, 2007, 20]. The second concern relates to the structure and topology required to enable collaboration. If government comparisons brought out features of centrality, so far, this work advocates the argument for a network topology that offers a distributed agency. This position, represented in the socio-technical network, is by nature, chaotic and changing and is not compatible with some cooperation reports, where strategies demand stability and non-acceptance of "mutant" elements, as presented by, for example, Axelrod [Axelrod, 2006, 56]. On the contrary, an unstable technology, such as software, thrives on mutant infrastructures, such as that provided by the collective, that is, the arrangement of actants represented by technical infrastructure and a participating community. As the history of FLOSS has shown, flexible infrastructure –mainly networks– and flexible communities of developers have produced usable software, demonstrating the possibility of achieving the approach. In summary, collaboration is a special type of cooperation that does not depend on central structures and, therefore, it has conveniently been adapted to the open software mode of production. In doing so, modes of decentralized organization are performed by the community, giving the whole socio-technical network implied the necessary levels of agency to obtain a valuable technological object within a software culture.

The importance of communication has already been stated in the previous chapter, and despite the Actor-Network Theory's shortcomings in its assessment, its prominence in collaboration should be apparent. However, communication is also important because language, technologically speaking,

is a technology. It plays a prominent role in the modes of software production because software is a language-based technology (written code), and natural written language is a key part of different communicative artifacts devoted to organization, goal discussions, and task coordination. Lovink depicts the intimacy of this relationship with the statement "[o]ur language changes through the technology that facilitates its transmission" [Lovink and Scholz, 2007, 19]. Similarly, Rheingold links the emergence of symbols (and writing) with cooperation. He further points to the symbiotic relationship between new forms of communication and new forms of collective action [Rheingold, 2007, 30]. With FLOSS, in particular, and developing the last chapter, computer networks, and today's Internet is at the center of the communication infrastructure and protocols implemented in the modern open modes of software production. The communication revolution brought about by the Internet as a medium lies in its ability to connect geographically dispersed individuals with the proper skills to become team members of a software project [Lovink and Scholz, 2007, 23]. And although, for example, participants with greater geographical proximity tend to exchange e-mails more frequently [Lovink and Scholz, 2007, 18], it can be said that its topology encourages collaboration without the need to design a proper collaboration process in detail [Rheingold, 2007, 59]. According to Rheingold, "[f]eedback loops can be as simple as establishing frequent e-mail communications or setting up team or enterprise wikis or blogs" [Rheingold, 2007, 61]. In other words, this implies what has been sustained before, and free software and open-source software movements have shown that network infrastructure and communication promote a *Laissez-faire* model of collaboration that cannot be planned beforehand.

Similarly, that a hands-on ethics or *getting things done* approach thrives in creating new social networks that operate as the new assembly line [Lovink and Scholz, 2007, 25], a statement that establishes a continuity with the hacker ethics tropes presented previously. However, despite establishing the relevance of the organization and communication, which will be addressed in later sections, one question remains concerning the motivation that moves participants to collaborate and become actants of the socio-technical network deployed in open software production modes. Some approaches, somewhat romantically, build on the motivations of the hacker culture and insist that the main source of the workforce in FLOSS is voluntary work [Gutowski, 2014, 5][2]. However, it has also been noted that this "voluntary work" is

[2]Even people involved in FLOSS are classified in this reference, for example, Richard Stallman is called "the idealist", Linus Torvalds, "the maker", and Eric Raymond, "the pragmatic", which points to the diversity of perspectives.

not merely altruistic; it can also increase the value of a coder in the market [Gutowski, 2014, 80]. Mainly at the turn of the century, other approaches based on the success of Open Source have argued that modern open-source software development is paid mainly by corporations that benefit from it. Take, for example, 80% of the Linux Kernel [Corbet and Kroah-Hartman, 2016, 12]; this offers an alternative narrative on collaboration. Another existing trend is collaboration in software as a gift economy, a trend that disregards whether the software production mode is based on voluntary or paid work. This movement replicates the connection of modern technology with ancients practices. Although appealing and, therefore, widespread, it has been contested because of the lack of evidence [Gutowski, 2014, 7].

Furthermore, the understanding of its reciprocity and to whom the gift is intended is unclear in the digital realm [Gutowski, 2014, 85]. There are others like Marthaller that represent the middle ground; they state that despite the similarities of the gift economy with open source, the latter is seldom antithetical to capitalism [Marthaller, 2017, 6,7]. To conclude, the role of politics and organization must be reiterated in the deployment of the community that takes part in open modes of software production, where collaboration takes place thanks to computer networks. This topic will be explicitly addressed in a section dedicated to teamwork in FLOSS.

5.3 The distributed code object

So far, software has been discussed as a complex technical artifact. Since its emergence, it has shown several traits of collaboration, from the early code-sharing societies in the fifties to the modern distributed production mode based on a socio-technical network encompassing communities and artifacts. In this sense, the network metaphor is part of this complexity, as pointed out by the popular reference to the so-called *network society*, in two distinct levels. The first is the technical infrastructure provided by computer networks and their potential for collaboration and communication. The second layer is a distributed socio-technical topology in which the relationships between human and non-humans are constantly negotiated –globally and in real-time –in modern software production. The alleged instability of software is also affected and even increased by the possibilities of the intertwined network. "[Software] is at once thoroughly pervaded by interlinked global standards and conventions (such as communication protocols), and at the same time is anarchically polymorphic and mutable" [Mackenzie, 2005, 72]. This statement somehow resonates simultaneously with the coordination requirements

of distributed undertakings and the social analogy of the bazaar. Software requires the rotation of several artifacts –mostly textual– the most notorious, perhaps, code. Therefore, the rhetoric of software's equivalence to code, a notion that has been contested, is popular among practitioners and some development methodologies.

This work has argued for a more complete definition of software, where code is part of a set of various artifacts, such as documents, diagrams, designs, and reports, among others, whose interaction persistently bring about the notion of a computer application as constantly developing process and a technological artifact that warrants understanding. Similarly, the socio-technical network responsible for the open and collaborative mode of software production constantly circulates these artifacts through distributed operations of read, write, and execution. It is precisely this last observation that provides code with such a special aura. Although most of the artifacts have their own performativity and agency, for example, a design can be read, but it is mostly created to be implemented, only code has the property to be read, discussed, written, and run by humans and non-humans and, in doing so, the purpose of software is achieved. This is the motivation to explore code as an artifact that is circulated in an incremental mode by other socio-technical artifacts built around it. This concern is embodied by the concept of the distributed code object.

As a result of the already established circulation of software –a condition that enables open collaboration– it can be said that modern software, and therefore code, is produced in distributed computer networks. However, as observed by Mackenzie, code is not only written or executed upon; it is "the outcome of complex interactions involving commodity production, organisational life, technoscientific knowledges and enterprises, the organisation of work, manifold identities and geopolitical-technological zones of contest" [Mackenzie, 2003, 3]. This statement indicates that code is a strong socio-technical artifact that encompasses several discourses beyond the exclusively linguistic arguments that are often used to state its materiality. However, code's linguistic and social dimensions are necessary to understand its preponderant role among the textual artifacts of modern software production. As mentioned, what first differentiates code from other textual artifacts used in distributed software production and intended mainly for communications (Wikis, mail lists, forums, FAQs, and web pages) is its agency[3]. According to

[3]Several references to "collective agency" were found to address the collaborative and social affordances of code. However, here, "distributed agency" is preferred because it fits

Latour, these textual machines lie between object and sign, thus their materiality [Latour, 1996, 222], indicating that text and things interact through the delegation of these executions regardless of whether the executors are human or non-human [Latour, 1996, 223]. The second difference is that the additional linguistic substrate of machines, such as code, is always permeating their operation. Mackenzie reminds us that "computer code never actually exists or operates apart from a prior set of practices, which allows it to do things" [Mackenzie, 2005, 76]. He notes that this enactment of the code as a textual machine is enhanced and supported by the cultural environment that surrounds the code. Thus, the concept of the distributed code object is raised to indicate that collaborative and open modern forms of software production are distributed, and code is a technological object operated by actants (human and non-humans), which simultaneously trigger some of their actions; lastly, that this double condition not only reflects agency but a performative character that is based on a set of cultural agreements. Ultimately, the so-called version control systems becomes an indispensable artifact to observe these negotiations.

This artifact, the version control system (sometimes called a revision control system), "is a combination of technologies and practices for tracking and controlling changes to a project's files, in particular, to source code, documentation, and web pages" [Fogel, 2017, 52]. The complex nature of this artifact surfaces clearly from this definition; it articulates, as mentioned, a technological dimension alongside a cultural one. Version control systems mainly affect texts (the mentioned textual machines), which are imbued in the constitution of software (code, documentation), as well as the communicative requirements of the socio-technical network set in motion for its open collaborative development. However, this artifact's broad scope is code, and it is especially the textual and performative element of software that benefits the most from version control systems (VCS). There are two main reasons for this observation, which can be intuitively recognized within the development community and how intensely this artifact is used. The first reason is that agency and performance are properties shared between code and VCS. Many VCS rely on the classic UNIX command, "diff", to make simple textual comparisons between files [Fuller et al., 2017, 97] (a linguistic dimension) and build a set of tools whose functionality is deeply integrated into the manipulations of the distributed code object (human and non-human), and the corresponding behaviors displayed by human programmers (a cultural dimension).

the socio-technical network concept better and can have both interpretations of distributed actants, human or non-human.

The second is that most of the normal tasks accomplished through VCS are code-related; "[aspects of a project supported by VCS] inter-developer communications, release management, bug management, code stability and experimental development efforts, and attribution and authorization of changes by particular developers" [Fogel, 2017, 52,53]. Although communicational artifacts are heavily circulated and gradually modified in a distributed manner through VCS, it is the circulation of the distributed code object that is the feature behind the emergence and evolution of open and collaborative software development in the times of the Internet.

The previous does not imply that other kinds of artifacts are not processed. However, software production's open and collaborative mode tacitly establishes a series of levels, where changing and circulating communication artifacts is the entry-level, and modifying and distributing code is advised for experienced users [Fogel, 2017, 57]. In sum, VCS are artifacts that support a code-centric approach to open software development; this can be attested by the connection to UNIX's modular approaches [Fuller et al., 2017, 89]. The GitHub, which was the most popular VCS at the time of this writing, was based on the Git, created by Linus Torvalds in 2005 [Fuller et al., 2017, 87] to perform Linux Kernel code endeavors. However, the possibilities of the VCS go beyond UNIX's "crafted, knowable code with powerful abstractions", so important to the evolution of the Internet and FLOSS [Fuller et al., 2017, 97]. It provides a new mindset from where the old free software or open-source software discussions were overcome by a form of post-FLOSS development [Fuller et al., 2017, 96,97].

If these version control systems are one of the key innovations responsible in part for this alleged post-FLOSS mode of production, what then are their specific characteristics and importance within this socio-technical entanglement? What are the consequences in terms of the organizational behavior of the participants, the mutations of the technology, and, in general, the cultural changes undergone by the distributed code object because of the VCS? To address these concerns, we must first gain a better understating of the VCS. As mentioned, these systems basically work by using text comparisons to detect changes in files (in this case, source code files) and manage the evolution of the code base of a computer application. However, the evolution of collaboration on networks has also altered these systems. The first systems, such as the once-popular Concurrent Version System - "CVS" or "Subversion", were centralized; this granted writing access to the codebase to only a core group of developers [Rodríguez-Bustos and Aponte, 2012, 36]. Thus, software artifacts were located in a single server. The potential collab-

orator had to travel the learning curve to become part of the core group and contribute. Later, Distributed Version Control Systems (DVCS), such as the Mercurial or the already mentioned Git, appeared. They offered new ways to control the version of several software artifacts. The DVCS no longer relied on a single server or repository. A developer could clone an entire project into his/her personal station and work on software artifacts, even off-line, make (commit) changes and, when convenient, send them to a group of actants, such as automatized collision detectors and human reviewers, that would decide whether to include the contributions in the main project body, thus conserving its integrity. In this sense, DVCS "allow teams to easily create, implement or switch between different workflow models. Thus, the classic centralized model is no longer the only option" [Rodríguez-Bustos and Aponte, 2012, 37]. As will be addressed in a later section, open collaborative production modes, particularly FLOSS, have had different team organization schemes. Despite the notable flattening potential of open collaboration, sometimes teams are organized hierarchically. For instance, the Linux operating system, and the tension between representing a "collective agency in the process of constituting itself" (a peer-to-peer potential) with being a "highly centralized project in many ways" [Mackenzie, 2005, 73]. Among DVCS's improvements to open collaboration is a more inclusive process that encourages participation and implements the corresponding contributions more smoothly. Having established this simple differentiation within the versioning technology, it is possible to return to the features of the modern version system, namely, the DVCS. According to Fogel [Fogel, 2017], these are some of the main operations and concepts implemented in these kind of systems[4]:

- *Commit*: This is possibly the most used operation; it refers to a change made on the code base. In a DVCS, the change can be published immediately. The DVCS's architecture allows modifications on local copies of a project, and every developer can hosts his/her own repository. On the other hand, the centralized model collects this changes in a single place.

- *Push*: This is, basically, publish a (commit) to a public repository. In centralized systems, it is a common practice to send the commits to a single central and official repository. In the distributed model of DVCS, the developer can choose where to send his/her changes, flattening the relationship between core developers and sporadic contributors.

[4]There are more operators and constructs in VCS, whether centralized or decentralized. However, their description is outside of the scope of this work in which only those concepts required for the argumentation are mentioned.

- *Pull*: This is used to pull the changes (commits) that other developers have pushed into one's personal copy of the project. Because VCS allow working various versions (branches) of a project, when the pull is made on the main branch (the origin of the official version of a program) it can be considered as an update.

- *Repository*: This is one of the key projects of all the version management philosophy. The repository is a database where changes are stored to be published later. The centralized model relies on one central repository where all changes are sent to and obtained from. Logically, in the decentralized model, every developer can have his/her repository; this creates a problem when trying to determine which one is the official master repository. This issue is solved through negotiation and social agreements and not by a technical choice, as in the centralized model.

- *Clone*: This is the simple operation of obtaining a local copy of code base and working on it.

- *Branch*: This is literally a line of development, that is, a copy of the project isolated from others co-existing in the same version managing system. Changes made on one branch does not affect the other branches unless ordered so using "merge". This operation allows different development steps with different objectives to happen at the same time, resulting, for example, in the common feature of FLOSS projects, which have an experimental version and a stable version of the software.

- *Merge*: This operation can be done in two ways. Changes can be moved from one branch to another; for example, to include new features from the experimental branch into the stable branch, or to merge changes made by different developers to the same file but in different parts, that is, non-conflicting changes.

- *Conflict*: This is when two or more developers change the code in the same place. This is an example of actant negotiations; the version system will notify the problem, but it is up to a human to resolve it.

- *Reversion*: This is the operation of undoing a change made to the software; it is also an example of actant negotiations. A reversion is stored as a change, the system must revert to its previous state instead of allowing the file or part of the software to be manually modified and submitted again.

- *Lock*: This secures a part of the software from further modifications. It is a conservative approach to prevent conflicts. It is normally enforced by a centralized VCS in what is known as "lock-modify-unlock". Conversely, the decentralized VCS promotes open collaboration where no locking is required in order to perform a commit, thus bearing a more horizontal approach called "copy-modify-merge".

To advance to a conclusion, the socio-technical network responsible for open and collaborative software modes of production today would be inconceivable without the circulation and modification of the distributed code object. Artifacts, such as DVCS, play a huge role in the development of new forms of networked collaboration, allegedly post-FLOSS, between the collective. The collective referred to is constituted by non-human actants based on linguistic operations, mainly differences in the source code or communicational artifacts, in general, and human actants with a set of cultural rules to negotiate the incremental development of software based on geographically distributed contributions. Additionally, the granularity and dispersion of the code object on the network maintain and even accelerates the unstable nature of software. However, it simultaneously manages to produce practical and usable results. Software not only exists in multiple instances and versions scattered in repositories and the local hard drives of the developers, but it also exists in several temporal stages with distinct levels of evolution coexisting together. FLOSS's common practice of having a stable and experimental line of development points precisely to the ever-present possibility of forking; that is, taking out a specific distributed code object to start an entirely new development, mostly for philosophical reasons among developers.

In this train of thought, two characteristics promoted by this model are paramount to grasp modern software production, changes and branching. As Fogel notes, the core of the VCS, in general, is a management change, that is, "identifying each discrete change made to the project's files, annotating each change with metadata like the change's date and author, and then replaying these facts to whoever asks, in whatever way they ask. It is a communications mechanism where a change is the basic unit of information" [Fogel, 2017, 53]. This discretization in changes enabled the flow of the distributed code object in computer networks. The nodes of these collaboration networks in a project are diverse. They mainly comprise repositories of publicly published versions and working nodes, such as DVCS, which include decentralized personal nodes of collaborators. The openness to available information on changes and the changes themselves not only imply the constitution of the referred communication mechanism but also a widening of

the collaborator base. The ability to have a copy of a project via the "clone" command, for instance, encourages this trait. Of course, this opens up the possibility for an increase in conflicts, however, the combination of automatic detection tools and human decision has apparently worked; this is evident in the fact that the copy-modify-merge model (as opposed to the lock-modify-unlock model) is the most widely used FLOSS development model [Fogel, 2017, 56]. This second feature operates as a corollary of the granularity of changing software artifacts and their transactions. Branching is a trademark of FLOSS development. It is very common for a project to have at least two lines of development, the stable and the experimental. Branching, not to be confused with forking, which consists of splitting a different project out of a codebase for different reasons, allows different versions to coexists, fostering innovation without affecting productivity. In this case, branching resonates with the hacker ethics and hands-on imperative, offering a testbed for new features to be implemented securely and collectively, and perhaps more importantly, potentially by collaborators outside the code group of developers.

To conclude it can be said that the possibilities of the distributed code object and its evolution, alongside the DVCS, are intertwined with the evolution of open and collaborative modes of software production in general, in particular, with FLOSS. The collective agency of the actants and software, in general, is performed continually through the circulation of changes in the codebase. This ongoing transmission indeed constitutes a "culture of circulation", that is, "a cultural process with its own forms of abstraction, evaluation, and constraint, which are created by the interactions between specific types of circulating forms and the interpretive communities built around them" [Lee and LiPuma, 2002, 92]. This reenacts Latour's dictum of linking one's tasks to a neighbor [Latour, 1996, 217]. In this case, to a networked neighbor.

5.4 Communication artifacts

By now, the paramount relevance of communications in software development efforts should be apparent, especially in the case of the open collaboration afforded by the kind of software represented by FLOSS. As previously discussed, FLOSS's open mode of production relies heavily on the communications capabilities provided by computer networks, that is, the Internet. More specifically, and following the concepts and vocabulary proposed in this work, the socio-technical network set into action in the development of modern open software requires permanent communication between nodes/actants in which various forms of agency must be observed by human and machine

actants to enable a successful exchange. Fogel observed that "[g]ood information management is what prevents open-source projects from collapsing under the weight of Brooks' Law" [Fogel, 2017, 37], referring to communication as a set of tools, protocols, agreements, and cultures dedicated to the exchange of information required to achieve a specific goal –a functional piece of software. This statement indirectly acknowledges the extreme importance of communication; that is, how information related to a project is generated, shared, stored, and retrieved in an open source project. However, it also problematizes it from the perspective of one of the most recognized software engineering's tropes, Brook's law. Fred Brook's observation, contained in the widely quoted "The Mythical Man-Month" (1975), states the inconvenience of adding more personnel to an overdue project, which generally results in an effect contrary to what is desired; in other words, the project will be delayed further. One of the reasons supporting this claim is the communication overload produced by the new personnel. This claim is noteworthy because, although Brooks' Law is referring to overdue projects, it raises a valid question.

How can communications cope with a possible overload, considering that, unlike Brook's examples from the 1960s, modern modes of software production employ a large number of globally distributed stakeholders sometimes interacting in real-time? The answer, which was implied above and in previous sections, is the automation of communications using software; this entails a distinct disposition of the actants. Therefore, if the socio-technical network of software production contains its own combination of technical platforms and related policies of conduct, which result in an effective communication and effectiveness is bound to the release of a piece of software, then "[. . .] the whole endeavor [communication] ultimately depends on sophisticated software support. As much as possible, the communications media themselves should do the routing, labeling, and recording, and should make the information available to humans in the most convenient way possible. In practice, of course, humans will still need to intervene at many points in the process, and it's important that the software make such interventions convenient too. But in general, if the humans take care to label and route information accurately on its first entry into the system, then the software should be configured to make as much use of that metadata as possible" [Fogel, 2017, 37]. Following this observation, it can be argued that modern software projects based on networking not only depend on communication among the community (users and developers) but also on the close coordination of humans and specific software platforms used for communication. In doing so, not only is Brooks' Law somewhat refuted, but collaboration between several parts is

also achieved.

Parting from the premise that communications in modern software modes of production are complex and how the network metaphor and software-human negotiations account for the possible increase of interactions produced by more participants (actants), characterizing these computer networks enabled communications becomes necessary to pursue a further discussion, particularly in the case of collaboration in FLOSS. This focus poses an important concern. Obviously, proprietary software has also benefited from the distributed software construction through a phenomenon called GSD or Global Software Development [Yu et al., 2011, 218], which resorts to the mentioned human-software negotiated communications scheme. Thus, what are the particular features of FLOSS when coordinating global efforts required to produce software? In this direction, the distinctions made by Taubert [Taubert, 2006, 44,45] become useful, with some adjustment, to proceed to the assessment of communication in the case of FLOSS. Briefly, his thesis is that this kind of software is organized differently from other types of software because the following three assumptions are met. First, its production mode is based on a network of heterogeneous actors (users, core developers, and sporadic developers), each actor has the ability to decide when to take part of this non-exclusive network. Second, no central coordination is required, each participant has the right to modify source code, contribute to it, or make changes that were not originally planned, even opposed to the original programmer's intentions. Third, electronic media, such as networks, are used to collaborate; face-to-face meetings are exceptional.

Although these three assumptions (particularly the last one) suggest communication's role in FLOSS development, as mentioned, they require some discussion. From the perspective of this work, on the one hand, considering only human actors is insufficient; on the other, the freedom granted to each contributor to change source code depends on the project's established policy and team agreements. Having said that, communications in FLOSS are further characterized and further explained through the following considerations. First, as hinted, communications in software in general, and, particularly in FLOSS, use the socio-technical platform provided by the Internet as a basis. Thus, there is a technical substrate with its related protocols, a set of users exchanging information on this technical substrate, and a set of rules and agreements established to allow effective communication. However, the complexity of the network topology and the global software development cannot be distinctly interpreted through the lens of classical communications theory of channel, message, transmitter, and re-

ceptor. Instead, automation and delegation of some communications duties on software require constant negotiation between the affordances of this technological substrate and the community using them. A feedback loop is, thus, established between the parts of the socio-technical network in which technical protocols shape the users' behavior and, in turn, user requirements can inform new platform features. Second, communications in FLOSS are mostly based on text. Because actants may be globally distributed and geographically separated, specifically human actants, such as users, developers, documenters, etc., communications are highly mediated and not conducted physically. Even though currently, the Internet offers various enriched alternatives, such as voice and video, text is still preferred because it can be easily archived, versioned, and searched; moreover, it uses low bandwidth. In this sense, text communications, for example, an e-mail, present similar traits to source code in a complex text condition. On the one hand, source code also has communication features that can be read by humans; on the other, it is intended to be read and executed by a machine (algorithmic agency). Text flowing through the socio-technical network articulates software and actors in communications schemes (distributed agency) and further operations, for instance, data mining, to obtain information to improve development processes. In fact, this corpus of communication, used as a project memory, has informed various FLOSS software engineering studies.

Regarding the latter observation, communications in FLOSS projects are usually open, meaning that they are accessible even to people outside a project who can progressively make part of the community. This feature confirms software studies' statements that FLOSS stands out as an object of study given its open nature and the availability of its communications. It also highlights how the socio-technical network of a project feeds itself from its archived communications, creating a knowledge base that allows the detection of communication patterns and improves the entire process.

From the previous sections, we can draw that FLOSS mainly employs text-based communications, and text is subject to the management of version control systems; this is usually the case with some documents and web site contents, among others. Here, the description of these distributed communication forms is continued. Four platforms and/or media have been selected to understand further the inner dynamics of the articulation between the actants in a FLOSS project. The following is a basic classification of synchronous (real-time) and asynchronous communication media, as Grassmuck suggests when describing FLOSS projects [Grassmuck, 2004]. The communication media selected are e-mail, more precisely, mailing lists, wikis, IRC,

and issue trackers, which are mainly used for bug management.

- *Mailing lists*: The e-mail and particularly the mailing list have been connected historically to FLOSS since the beginning. Not only did the GNU project (free software) and the early Linux Operating System post their first summons to collaborate on e-mail-based USENET forums but also, in the latter case, e-mail was the coordination tool for the Linux kernel development, before the version control systems were the norm. This statement indicates the many features of mailing lists[5] and the negotiations that its use implies. According to Grassmuck, e-mail and, more precisely, mailing lists are the central communication mediums for worldwide distributed collaboration in FLOSS; hence, its importance [Grassmuck, 2004, 241]. Mailing lists also function as a type of community log for a project, storing messages for years and creating an archive that provides context for new users and developers that want to join a project [Fogel, 2017, 50]. The automation provided by software expands the communication and collaboration between human actants. Activities, such as committing code into a DVCS or reporting a bug on an issue tracker system, will automatically generate and send the respective e-mail notifications to the interested parts. Spam detection is another mailing list communication activity that can be allotted to computer programs. Conversely, human control is required for highly social activities rather than enforced negotiation, such as moderation. In this case, new postings and messages from non-registered users must be approved. In both cases, human and non-human decision-making indicates the highly intertwined nature of the actants in the socio-technical network in task coordination; thus, communication. It also manifests the constant strive for openness in FLOSS, establishing policies that enable open participation. The obvious consequence is, as Fogel suggests that an entire culture is created around this mechanism, including the selection of a proper system, spam detection policies, filtering, subscriber management, moderation, and addresses obfuscation, among others [Fogel, 2017, 45]. In brief, communication development and decision processes take place in the digital and text-based medium of mailing lists; this allows individuals to collaborate in a common (software) project [Taubert, 2006, 45].

[5] Although reference was made to a forum and not a mailing list, as Fogel states, the difference between mailing lists and web forums is disappearing [Fogel, 2017, 42]. The difference depends more on the preferred access client used by the stakeholder, whether e-mail clients, web interface, or integral project hosting sites (canned hosting), such as GitHub.

"Many of these email mailing lists are publicly-viewable and archived for the long term, since FLOSS development relies on a certain level of openness in participation, transparency in decision making, and institutional memory" [Squire and Smith, 2015, 45]. Ultimately, messages and users are heterogeneous. Long text artifacts, such as reviews, bug reports, code snippets, error logs are sent thorough e-mail lists [Squire and Smith, 2015, 45], as well as design rationales by a community of participants mainly composed of core developers, a small group of them contributing most of the messages. Empirical studies, like Shibab et al., have suggested, through mailing lists archive mining [Shihab et al., 2010], that 10% of the dominant group produces 60% of the messages; this shows a clear correlation with a peak in mailing lists activity and code metrics, that is, source code changes [Shihab et al., 2010, 95,97].

- *The wiki*: Wiki pages are the most accessible entry to a FLOSS project for most of its users [Fogel, 2017, 69]. Its ability to be collectively edited resonates with the open nature of FLOSS. Ward Cunningham, its inventor, proposed the then (1995) new medium of the webpage as a solution for the loss of older messages in an early incarnation of mailing list collaborative software called LISTSERV [Boulton, 2014, 141]. As in other cases of inventions turned into standards by making them open, "Cunningham gave away the source code, and wikis became common in the programming community" [Boulton, 2014, 141]. Confirming this importance for programming, the first wiki was dedicated to the design pattern community [Cunningham, 2014]. Currently, as a text-based artifact, wikis are more associated with distributed and collectively managed editing projects. Perhaps, for this reason, wikis are understood in a software project as documentation, providing a better alternative to DocBook, for example [Hernández et al., 2008, 122]. However, its uses in software development encompass, of course, documentation, collaborative requirements engineering, and development environments [Hadar et al., 2016, 530], to name some applications. As noted, the nature of wikis is compatible with the FLOSS networked ethos and, indeed, enforces the FLOSS "community governance mechanisms", building on the famous "scratching an itch" FLOSS developing metaphor [Sowe and Zettsu, 2013, 16:4]. Specifically, from a communication perspective, wikis afford certain features well-suited for team communication because of their ability to make data visible to all, always on-line (available), and publishing changes automatically as they occur [Hadar et al., 2016, 532]. From a more structural perspective, wikis preserve the instability and complex nature of textual artifacts presented by different

types of agencies, whose purpose is to be read and written by a distributed community, but also executed to present the text through web pages. In doing so, wikis, like version control systems, deepen the long UNIX tradition and its philosophy of employing modular plain text artifacts that can be versioned [Hernández et al., 2008, 120]. Wikis also require a close interaction with humans to contribute content using plain text, which has a specific syntax that includes content and formatting elements, as well as wiki engines that use that particular syntax to present the content in hypertext form. Additionally, and taking advantage of the possibility of automating plain text, wiki engines also track changes and their corresponding authors, sending e-mail notifications and registering possible content-related disputes. Wikis are an important communication node in the socio-technical network, required to promote open collaboration and FLOSS as a product. Also, as a type of document, they attempt to use multiple resources (images, hypertext, typographic cues) to address a heterogeneous set of documentation artifacts present in a software development project, which includes text, source code, diagrams, abstractions, and images [Aguiar and David, 2005, 2]. In that sense, wikis also represent a negotiation field where not only human-machine conventions must be agreed upon but also cultural issues must be considered [Hadar et al., 2016, 530] in preparation for the coordination and communication required for effective collaboration in a FLOSS project.

- *IRC*: Perhaps the most recognizable synchronous communication medium in the FLOSS software development community is the Internet Relay Chat protocol, usually abbreviated as IRC. Noticeably, this protocol has survived for more than thirty years in a USENET environment since it was introduced in 1988, and although it might seem anachronic compared to newer alternatives. As Sinha, as well as others have noted in empirical studies of the Ubuntu Operating System community, the IRC has become popular and it is still widely used [Sinha and Rajasingh, 2014, 916]. One possible explanation is the IRC's open nature, not only as an open protocol with a corresponding open software implementation but as a communication platform that requires no registration. In fact, the open-source site, in its introduction to IRC, states, "[i]t is open to anyone. It is loggable. It is scriptable. It is discoverable. It is ubiquitous. It is free, in all senses of the word" [Brasseur, 2016]. This is why, despite the declining popularity of the IRC in real-time text communications, when compared to social network alternatives, it is still favored by the community participating in the

socio-technical network, responsible for modern open and collaborative software production modes. The IRC negotiates the communication of a diverse geographically dispersed community with different cultural backgrounds in a way that resembles face-to-face interactions using text. In turn, these IRC-facilitated interactions are multi-participant, dynamically changing, and question-answer driven, featuring a clique structure rather than a star network [Sinha and Rajasingh, 2014, 921] that raises the social nature of communication over the centralized network topology. The IRC offers the possibility of opening several channels of discussion, with a style of conversation that, in comparison to other media, encourages participation. However, because conversations can be large archiving can become problematic [Fogel, 2017, 69]; this can sometimes be perceived as a disadvantage considering that, as Alkadhi reports, the fundamentals of a software project are discussed on different media, including IRC. At this point, potential issues arise, given that some open-source communities disregard the discussion of detailed designs, which creates a lack of clear design communication within FLOSS projects [Alkadhi et al., 2018, 357]. Empirical studies show that design choices and rationale are discussed in 25% of IRC conversations; the figure reaches 54% when the conversations are conducted by code committers [Alkadhi et al., 2018, 357]. Finally, and returning to IRC's open properties, one of its features stands out, it is scriptable. The previous means that bots can automate some parts of the conversations [Fogel, 2017, 67]; for example, welcoming messages, forwarding conversations based on keywords, communicating with versioning control systems, and bug trackers. Once again, the point of actant communications within the socio-technical network dedicated to FLOSS collaboration is upheld, where other human-machine communications benefit from an open model of software production.

- *Issue Trackers*: Historically, software projects focused more on bug tracking tasks. This effort has evolved and expanded into tracking issues in general, such as new features, one-off tasks, and other activities that must be openly managed [Fogel, 2017, 62]. Issue tracker systems are a fundamental part of the communications infrastructure in a FLOSS project. They provide a channel for normal users to report bugs and request new features, as well as for the development team to diagnose, manage, and inform on the progress of such reports. As is the case with other communications processes described previously, an issue tracker system seldom operates in isolation. Instead, it is connected to e-mail and wikis to forward requests through different

> channels automatically. Issue tracking is yet another case of human-machine communication, in which actants collaborate in the incremental FLOSS development process. Although issues trackers are generally dedicated, bug tracking remains the most recognizable case of this type of platforms. Specifically, they indicate the communication challenges between actants. On the one hand, bug tracking systems also communicate through other channels, with some degree of automation. On the other, bug tracking is a highly social task. It requires the participation of a community to report issues, which overlaps with a validating community that detects false bugs or poorly documented cases and a developing community, in charge of fixing the remaining bugs. This clearly requires a collaborative life cycle between these different communities and demands close communication between various nodes (actants) of the socio-technical network. In this regard, one aspect must be stressed in bug management, that is the very empirical nature of the entire process. As Taubert maintains, there are experimental traits in bug management. While describing the feedback loop implied in the bug management life cycle, that is, that software evolves based in part on bug reports, this author expresses the difficulty of determining a bug in the FLOSS domain, as opposed to the proprietary and controlled model [Taubert, 2006, 187]; he describes bug management as a *Realexperiment* [Taubert, 2006, 188]. A *Realexperiment* requires a specific experimental framework, in this case, the bug must be reproduced and a certain execution environment must be provided. It differs from a normal scientific experiment because it emphasizes the importance of consensus and community agreement, prioritizing designs over knowledge generation [Taubert, 2006, 189]. To conclude, it is important to remember software's unstable nature and FLOSS's, in particular, which is embodied by a development process with no obvious conclusion that makes free software incomplete [Taubert, 2006, 188]. This particular trait is related to the bug management and communication processes widely present in the socio-technical network of open mode software production.

In summary, the selected communication artifacts show the constant exchange between different parts of a socio-technical network required in open and collaborative software production. They emphasize the close human-machine collaboration displayed by several levels of automation between these actants. Similarly, the textual nature of these artifacts is highlighted by pointing out that "[d]evelopers in OSS projects rely heavily on written communication to collaborate and coordinate their development activities

such as IRC channels, mailing lists, and wikis. Developers' communications occurring over these channels contain a wealth of information about the software system, such as design decisions" [Alkadhi et al., 2018, 357]. Reinforcing this point, the use of other textual artifacts, such as Pastebin, intended to fill the gap where code exchange can be problematic within a natural language conversation [Squire and Smith, 2015]. Therefore, terms such as "literate programming", coined originally by Donald Knuth, have been used to indicate that a piece of software can be understood as a piece of literature, addressed to humans (the community) rather than a computer [Xiao et al., 2007, 178]. This points out the relevance of social conventions intertwined with a communication practice. However, the textual nature of software and communications artifacts challenges a simple dual classification. They may operate as documentation or programs at different levels, depending on the automation of distinct stages of communication.

5.5 The distributed team: a complex community

By now, it has been repeatedly stated that the modern collaborative software production mode operates as a socio-technical network. It implies a complex assemblage in which a community (social part) and artifacts (technical part) are negotiated at different levels, following the affordances of the network paradigm, as a cultural metaphor entangled with a technical substrate, namely, computer networks. This observation has cued the approach to the nature of artifacts in previous sections, concluding that most of these artifacts are text-based; therefore, they are easily stored and transmitted electronically (source code being a special type). Similarly, that they allow various levels of automation, leveraging human agency with a dynamic lot of technical agencies in which the technical artifacts' algorithmic and interactive affordances entail a distributed agency. So, because we have already covered some software and communication artifacts that assist this collaborative mode of software production, the social factor will be briefly discussed in this section. This is not an easy task, as the social aspects of software production are challenging –and exciting– field of inquiry. In fact, as observed in Pabón et al., [Pabón et al., 2016], some authors have pointed out that research on software development process prioritizes technical aspects over human aspects. However, it can be said that this trend is changing. Ostensibly, this gap is deepened when considering FLOSS, given its particularities.

Although the popular images of the cathedral and the bazaar previously discussed can be perceived as dichotomic oversimplifications to describe FLOSS' uniqueness, it is undeniable that the open and collaborative production of software embodied by the FLOSS ethos represents several complexities when compared to traditional development processes. As Taubert explains, when characterizing FLOSS' development as an anarchist enterprise, three assumptions indicate the novelty of FLOSS development compared to other forms: 1. The actors are heterogeneous (different roles and cultural background) 2. The actors can take another's contribution and change it even though the modification does not reflect the original intention, but fulfills the project's objective, without needing a central authority, and 3. The actors are connected with the others mainly through electronic media (i.e., computer networks) and rarely meet face-to-face, except for special events, like conferences [Taubert, 2006, 44,45]. Although these statements require context and do not always hold true, or as in the third case, they are an exclusive FLOSS development practice; they show that it is the human component that makes FLOSS a special mode of production. In other words, although some level of automation is possible, it is the politics, organizational traits, motivations, and other human principals that define this type of software development efforts. Ultimately, the creation of a distributed work team must coordinate efforts, determine a curse path for the project, and coordinate the social network, assigning the tasks an roles in a seemingly non-hierarchical manner, as well as grant the technical and social potential to open and collaborative software modes of production. Thus, governance, organization, and motivation are briefly addressed as a way to shed light on these human tenets of the socio-technical network.

One of the contradictions at the core of the discussion on collaborative software development is that various practices, such as having a distributed team, non-hierarchical governance, and focusing on coding over detailed design can, from the perspective of software engineering, be seen as counterintuitive. However, empirical evidence and, mainly, the success of several FLOSS applications show that the open, collaborative approach works [Giaglis and Spinellis, 2012, 2627]. This points to not only the current technical tools that are commonly used in software development (open or not) but also to how the workforce is organized and directed. Socio-technical metaphors, such as "peopleware", coined in DeMarco and Lister's well-known organizational book of the same name [DeMarco and Lister, 2013] describe the entanglement of technical and human components and highlight the importance of communication. Although this terminology could be applied to any modern technical project, "it is very difficult to find a modern industry, if there is

one, where the degree of heterogeneity is as large as the one we can see in free software" [Hernández et al., 2008, 59]. Following this idea, it can be inferred that social and human capacities have always been complex in a collaborative project. However, distributed communications through computer networks pose various new challenges simultaneously, as well as new possibilities for common achievements. Specifically, that governance –meaning management and coordination– consensus and democracy among distributed software development teams is of paramount importance.

First, on governance, the standard rhetoric of horizontality in open, collaborative software production has several nuances that must be addressed. One widely used statement is that some projects run on what has been called a "benevolent dictatorship", meaning that one person is responsible for setting the tone of a project and coordinating its team and tasks. Clearly, this creates a tension between the centrality of this management mode and the distributed nature of open software development. However, as Fogel observed, this idea of a dictator does not hold, mainly because developers can always leave if they feel that their opinions and contributions are not properly appreciated [Fogel, 2017, 74]. Through closer inspection, it can be said that the power of a central figure can be relieved in two ways; one is merely social and the other, socio-technical. In Taubert's mentioned assessment of FLOSS principals, he observes the heterogeneity of its actors and how they contribute to a particular project, the actors are free to join and leave the project, ruling out the possibility of excesses and impositions from a dictator. The socio-technical factor is conspicuous when one of the fundamental conditions of modern FLOSS is considered: forkability. Version control systems, namely, their distributed variant, allow the programmer the technical ability to copy a project's entire code base that can be locally modified, this also brings with it the political possibility of working individually, without central control [Fogel, 2017, 24]. Negotiations are only required when committing changes. In this respect, Fogel insists that forkability operates as a possibility more so than as a commonly applied measure [Fogel, 2017, 73].

Furthermore, Fogel provides an intuitive classification that separates "soft forks" in which the customarily distributed copy is made, from "hard forks", where more serious issues among developers can cause a division –sometimes irreconcilable– within the project [Fogel, 2017, 189]. Ultimately, the benevolent dictator figure can be considered a definite exaggeration. Consider, for instance, the well-known case of the Linux operating system. Although the final decisions depend on one person –Linus Torvalds– the system continues to thrive after more than two decades. Collaborative projects tend to behave

in an organic, auto-organized way, given the nature of distributed communications and the network metaphor.

Second, in addition to project direction, consensus, and democracy, are understandably a key part of the entire collaborative socio-technical network required for software development. Politics, which, by definition, requires the participation of a group of people, cannot be avoided in FLOSS projects and collaborative projects in general [Fogel, 2017, 166]. Returning to the idea that the collective, which comprises a technical part and a community that makes up the workforce, it can be said that the issue with democracy, as is often the case, is that the FLOSS project community is heterogeneous. Its participants have different roles and backgrounds, something that will be addressed below. For starters, let us consider projects whose community adheres to a kind of constitution, as in the case of the Debian operating system, where an order of deliberation is clearly established [Debian Foundation, 2016]. Then, let us consider other projects, which have operated more chaotically, given the different times, the importance of certain roles within them (usually developers have more power of decision) and the fact that these communities have an onion-like structure, meaning that there is a core group and an external group [Wang and Perry, 2015, 119]. We can thus infer that members of the core group have more decision power. In either case, negotiations are integrated into technical discussions such as the right to repair a bug, add a feature, or how to document interfaces, which lead to more fluid agreements [Fogel, 2017, 74]. Once again, this suggests the hybrid nature of the socio-technical network, which involves automation, informal textual communications, and decision making that happens thanks to the technical affordances of computer networks and the distributed culture indebted to it. Automated and distributed version management through VCS is an example of this combination, allowing the flow and replication of different software artifacts. This is important because, as Fogel expressed, "[r]eplicability implies forkability; forkability implies consensus" [Fogel, 2017, 73]. This quote must be carefully assessed because it could imply a somewhat technical determinism surrounding a social construct such as consensus.

Nonetheless, human aspects remain decisive when considering on-line communications in which personality is projected linguistically; that is, it is perceived through language [Pabón et al., 2016, 9]. This statement, based on research mining effort in FLOSS repositories, suggests the socio-technical aspects of all development efforts, including the decision process. It restates the previously established importance of e-mail as a communication tool for dialogue. In sum, the socio-technical network of modern collaborative modes

of software production is diverse not only in its socio-technical entanglements (different levels of automation) but also in its community (the human part). This deepens the already complex social principles of the communities, each with global patterns of communications and potentially different backgrounds, as well as the work ethics of its members. Yet again, it is highlighted that governance, the settlement of debate, the management of group members, and project decisions are weighty features, especially in self-organizing systems, such as open, collaborative software production [Fogel, 2017, 73].

To understand the community of people who participate in a collaborative and distributive team, it is necessary to understand how this community is structured, organized, and how it operates. First, the roles of a distributed team are diverse. Although developers and some committers; that is, developers whose changes are incorporated into a codebase from a VCS, are usually perceived as the most important members, there are other roles and tasks (documentation, bug and issue management, translations, legal advisors, etc.) whose influence can surpass the developers', even committers [Fogel, 2017, 78]. Some of these roles are listed below, following Fogel's classification [Fogel, 2017, 177-184]:

- *Patch manager*: organizes the patches (updates or bug fixes) received from different channels (VCS or e-mail, for instance).
- *Translation manager*: usually does not translate himself/herself and rather manages contributed translations of documentation, code, or message errors.
- *Documentation manager*: supervises changes in documentation, a task that is not easy considering that documentation is usually the most accessible way to collaborate in a project.
- *Issue manager*: manages bug reports, follows up the developments, cleans old reports, and classifies groups of reports.
- *Transitions*: supervises which tasks are not accomplished and negotiates with the members of the group responsible.
- *Committers*: developers who submit (commit) code changes using VCS; usually, the changes are made first on their copy of the codebase. This is a complex role because of 1. Not all code maintainers are committers, 2. There are different levels of access to the code base, total or partial in certain modules, and 3. Committers usually wield more decision

power, as pointed out by the onion-like structure of these distributed teams.

This classification has been favored for its simplicity. However, there are other appraisals of the roles and their transformations (change from one role to another); some of them are briefly described in Wang et. al, [Wang and Perry, 2015, 119]. In this work, the authors note that these roles are dynamic and implicit; that is, a central project manager does not allocate them, instead, they are based on the quality of the contributions [Wang and Perry, 2015, 120]. In turn, the onion-like structure, which bears levels that make some roles more peripheral to others, provides participants the ability to change or increase their involvement. Along with the mentioned division of code (or other artifacts) into different groups with different access levels, it grants a type of "responsibility without monopoly" [Fogel, 2017, 177], which, once again, converges into a consensus among the community. Adding to the intricate setting of this distributed community, participants' personal traits evidently affect how software artifacts are handled [Pabón et al., 2016, 8]. In fact, the study by Paruma Pabon et. al, using data mining on specific e-mail communications, identified clusters of traits. Cooperation, sympathy, conscientiousness, achievement motivation, cautiousness, openness, adventurousness, imagination, intellect, liberalism, conservation, and self-enhancement were personal features that ranked high in clusters of communications among developers [Pabón et al., 2016, 12]. Ultimately, the appraisal of a community would benefit from approaching some demographic facts. In their exhaustive study on free software, Hernandez et al., have offered some noteworthy points concerning the heterogeneity of the FLOSS community, which coincides with the thinking of other authors, including Taubert. Some of their remarks are listed below [Hernández et al., 2008, 57-62].

- The heterogeneity of the community surpasses preconceptions that mythologize the hacker culture.
- Most of the developers of a FLOSS project are in their twenties.
- There is a strong bond between the FLOSS community and Universities.
- Despite the ongoing tensions between FLOSS and traditional software engineering, 33% of participants are males that define themselves as software engineers.

- 80% of the participants express that most of their contributions are made in their free time.

- Most of the participants are males and are located in the United States and Western Europe.

However, some precisions are required, given that the drafting of this study dates back to ten years. Although gender, ethnicity, and country of origin are still factors that may undermine the rhetoric of diversity, some changes have occurred. Most notably, the switch from voluntary work to development with mixed sources, where privately supported efforts play a huge role. This combination assumes the human side of the open software development socio-technical network as a workforce that operates around the clock and, hopefully, around the world.

Reprising the socio-technical network at work in open, collaborative software production involves a set of complex artifacts intertwined with social constructs whose coupled operation enables project management and decision making. The actors/nodes of this socio-technical network carry out several automated processes in the exchange of information –mostly textual– on a computer network layer; this enables the building of a social meta-layer of human negotiations. These exchanges of opinion are accomplished, on the one hand, by human actors that are mostly free to interact or not. On the other, they involve some temporal hierarchies that contest the standard rhetoric of FLOSS horizontality. However, the distributed teams participating in these endeavors must be motivated to keep these distributed organizations working collaboratively. Thus, motivation is an essential tenet in understanding the behavior of these socio-technical constructs. The motivations referred to imply what moves potential actors to join a particular project and how it stimulates them to continue. An early study on the matter by Lakhani et al. identified two kinds of motivations: intrinsic, related to fun and self-fulfillment and extrinsic, related to community obligation and sense of duty [Lakhani and Wolf, 2005, 3], creativity as the intrinsic driving motivation of several participants of their empirical study [Lakhani and Wolf, 2005, 5].

Noting that other factors, such as improving programming skills to increase reputation and access better job offers, remain major motivators, it can be said that motivation resonates with the pragmatic idealism that permeates several FLOSS practices. Fogel makes recommends that some practices should be observed to keep actors satisfied once in a work team. Delegating tasks to show trust, managing criticism diplomatically, and giving credit

(not only in the source code) [Fogel, 2017, 169] can be seen as practices to maintain the well-being of the core group. Likewise, recognizing the issue reporters' contributions and answering –when possible– communications enthusiastically can stimulate outside participants. In turn, the participants will consider taking on additional responsibilities because people that receive attention will maintain participation, even if their contributions are not always incorporated [Fogel, 2017, 178]. The lack of appropriate motivation will eventually lead to turnover, that is, the loss of participants. The previous is important because distributed groups are dynamic not only in composition (member numbers, roles, and turnover) but also in time. Some projects have a lifespan of several years , with a generally unstable core group, and obtaining information about a project is a great asset [Izquierdo-Cortazar et al., 2009, 2]. These distributed teams can also be seen as "virtual teams". This concept, coined by Gliaglis, challenges traditional thinking about distributed work. It notes that "virtual teams consist of highly skilled and motivated individuals, who leverage the power of communication technologies to overcome problems associated with physical distance" [Giaglis and Spinellis, 2012, 2625]. This observation highlights communications as the main currency for producing the collaborative and open features of FLOSS and its socio-technical network successful in harnessing the potential of 24-hour global collaboration.

5.6 Abstract artifacts

Until now, the socio-technical network employed in collaborative software development, with FLOSS as the most prominent example, has been described as an arrangement of interconnected actants/nodes, which are constantly in communication through the exchange of mainly textual artifacts, such as source code, documentation, and real-time and asynchronous messages. Software automation intertwined with human negotiations provides an iterative framework for software activities taking place over time. Historically, FLOSS has ostensibly privileged source code as its primary artifact (discussed in methodologies in this section). However, the importance of communicative artifacts in modern software production is undeniable. In fact, one particular characteristic of FLOSS is the tension between these artifacts, presented in previous chapters as the trade-off between a hands-on approach (coding as the principal activity) and planning (design). More precisely, communicative artifacts could be placed in the second group (design) because they reflect –direct or tacitly– the architectural discussion, shared rationales, and design decisions that go beyond the communicative agency of source code. The is-

sue of artifacts not merely part of software as a product or process; it scales to a higher level, where they are part of the larger picture of software as a culture. This statement brings up what will be referred to as meta-artifacts, a set of practices, guises, and their corresponding consideration as a system that supports the operation of the entire socio-technical network.

Clearly, these meta-artifacts interact with the other artifacts (code and communication), providing a kind of umbrella level where the relationship between the artifacts is organized. Three instances are considered, the licenses addressing the legal aspects of software and its process, the methodologies that suggest how the distributed workforce should proceed, and software design artifacts that deploy complex forms of communication beyond text to represent software architecture choices. It should be noted that these three meta-artifacts are complex and, in turn, are composed of several alternatives, and they deal simultaneously with textual communicative artifacts and source code, but on different levels. With that in mind, a brief historical overview of these meta-artifacts is provided to shed light on their historical and structural importance for the assembly of artifacts in the socio-technical network of open and collaborative software development.

5.6.1 Licensing

Intellectual property and its accomplishment through the issuance of licenses are key issues in understanding FLOSS. These encompass from popular cultural behaviors concerning the nonexistence of cost of a software product to complex legal infrastructures at a personal and corporate level. In general terms, the part of intellectual property most related to software is the copyright; this points to the nature of the copy inherited from the publishing world, which brings us to the first consideration. Usually, the discourse on software and copyright resorts to the *intangible nature* of the former, with FLOSS "advocating the benefits of copying and incremental innovation versus exclusive control of a work by its author" [Hernández et al., 2008, 41]. Although intellectual property issues emerged after the commodification of software in the late 1960s, the software field always had a hidden economy of software exchange [Berry, 2008, 109], which involved the copying of software artifacts, mainly code. As mentioned in Chapter Three, the history of software and copyright has had several milestones that shaped the modern idea of FLOSS; among them, IBM's unbundled software, UNIX's ambiguous license schemes, Bill Gates' letter addressed to the hobbyists, and the invention of the copyleft concept incorporated in the GPL's. The previous may constitute a clear example of Berry's observation that "modern conceptions

of property rights and commons-based ownership are the result of contingent moments in history rather than the outcome of rational, planned or natural trends"[Berry, 2008, 81]. In short, the evolution of open, collaborative software production modes displays a continuous tension with the concepts of property and the extent to which copying is exercised. It should be noted that regarding copyright, whether software should be considered as a tool or a literary work, the latter being favored, is still a matter of debate. This resonates with previously discussed ideas, such as the instrumental conception of software or its appraisal as an elaborate work with both positions postulating different and sometimes conflicting legal, economic, and ethical perspectives.

The intellectual property regime is not only a pragmatic principle of every software product, but it also expresses the particular ideology of the participants and constitutes one of the key elements of the socio-technical network placed at the service of open and collaborative software development. Regarding open source software, Weber posits that the objective of the intellectual property regime is "to maximize the ongoing use, growth, development, and distribution of free software" [Weber, 2004, 84]. Although this phrasing seems clear, the subject is, indeed, more complex. FLOSS licenses, in particular, are different from private software licenses, considering the specific terms under which the authors distribute their programs, giving rise to new methods of development [Hernández et al., 2008, 45]. In other words, from a legal perspective, these licenses enable the entire distributed agency by the work teams and the assemblage of software artifacts contained in the socio-technical network of open and collaborative software development. Although Weber's quote can be understood as a general aspiration shared by the different FLOSS trends, its implementation remains problematic and even produces incompatible licenses. Hernandez et al. provide a simple classification of FLOSS licenses. The first is "permissive licenses", which grant the user virtually any condition and freedom on the software, in addition to the rights of use and redistribution and even the possibility to close the source code and develop a private application. The second is "strong licenses", which aim to ensure that access to the source code is granted to all users, ensuring that the software always remain free [Hernández et al., 2008, 47-50]. Popular licenses, such as the BSD [The Open Source Initiative, nd] and Apache ApacheFoundation2004 can be found in the first group. The very well-known General Public License-GPL [Free Software Foundation, 2007], perhaps the most influential license in the FLOSS realm, and the Mozilla License [Mozilla Foundation, nd] are prominent examples of the second group.

It should be noted that the dual approach of this classification is the basis on which a particular community is favored; this sets the priorities. Permissive licenses favor the flexibility of the developers, and the general community (users) is paramount for strong licenses. An additional classification is provided in a study by Sen et al. in which licenses are connected to participant motivation. The categories it proposes are "Strong copyleft", "Weak-copyleft", and "non-copyleft" [Sen et al., 2008, 211]. The first and last options are assigned roughly to the classification mentioned previously; in this case, the GPL would be strongly copyleft, and the BSD would be entirely non-copyleft. This arrangement opens up a middle ground category (therefore weak) that includes alternatives, such as the Lesser General Public License (LGPL), which allows greater interaction with other licenses than the restrictive GPL. This classification shows the General Public License's legal success in defining other licenses and setting the tone for the intellectual property in FLOSS, despite the decline in its popularity from half of the licenses [Sen et al., 2008, 220] to about 25% in popular platforms, such as GitHub [Balter, 2015] and [Bacon, 2017].

As suggested, there is a link between license and participant motivation. Sen et al. established a relationship between intrinsic (problem-solving) and extrinsic (peer recognition) motivators and types of license. They linked the first group with moderate licenses and the second group with less restrictive licenses [Sen et al., 2008, 227]. Although this reasoning is presented as a hypothesis, it points to the tensions between open source software objectives; they state, "managers who want to attract a limited number of highly skilled programmers to their open source project should choose a restrictive OSS license. Similarly, managers of software projects for social programs could attract more developers by choosing a restrictive OSS license" [Sen et al., 2008, 208]. In part, this discussion is the reason behind the division in FLOSS between free software and open-source software. The latter is more business-oriented and, therefore, favors the freedom of the developers. The previous is noted because the classic FLOSS rhetoric discusses the advantages from the users' perspective, leaving out several considerations for the developers. Concerning a socio-technical network and the distributed participants of its associated work team, a few aspects involve intellectual property issues. For instance, licenses must be agreed upon, when a change of scheme is proposed, this poses some governance issues. Every iteration of a software product can opt potentially for a different license. The potential for forking brings another layer of complexity. Lastly, when participants leave a project, it must be clear what will be done and who will take care of their "orphaned code". These aspects show that modern modes of software

production must deal with this layer of complexity to provide the distributed agency required for collaboration while considering that the unstable nature of software permeates the legal realm.

5.6.2 Methodologies: agile methodologies

The question of whether FLOSS has its own software development methodology has been an ongoing issue since terms such as free software or open-source software were first coined. Similarly, and as argued throughout this work, the presence of collaboration and modes of software production, comparable to FLOSS in the history of software, requires a broader assessment of the same question. For starters, as Warsta and Abrahamsson note, there is a contradiction in FLOSS projects, as there are several examples of successful applications, yet, there does not seem to be a method for building them [Warsta and Abrahamsson, 2003, 1], at least from an orthodox point of view. They describe FLOSS in the following terms, "[It is] not a compilation of well-defined and published software development practices constituting an eloquent, written method. Instead, it is better described in terms of different licenses for software distribution and as a collaborative way of widely dispersed individuals to produce software with small and frequent increments" [Warsta and Abrahamsson, 2003, 2]. This definition provides a practical answer to the question, namely, that there is no method; it also restates the complex nature of distributed modes of software production. In the same line, Healy and Schussman point out, referring specifically to open source, that its approach "does not fit with standard models of software development" [Healy and Schussman, 2003, 2]. Moreover, the highly stratified nature of software projects carry terms, such as "OSS approach" or "OSS community" as generalizations [Healy and Schussman, 2003, 13].

This picture directly contradicts the popular rhetoric of the so-called "bazaar" approach, often used as a FLOSS trope, particularly in open source, to support the idea of a vernacular and computer network-aided method in collaborative software production. While imaginative and colorful, the bazaar metaphor is a very problematic oversimplification. Specifically, regarding the issue of a distinct FLOSS software development method. On the one hand, it can be said that the mere idea of the bazaar is misleading, because FLOSS software can be developed using any method [Hannebauer et al., 2010, 1]. On the other, the bazaar directly contradicts empirical research, making the distinction between the bazaar and cathedral, with the latter being the hierarchical antithesis of the former and supposedly preferred by close source projects. This claim is "illusory" in not acknowledging that "[t]here are

cathedrals in the open-source sphere and bazaars in the closed source" [Wilson and Aranda, 2011, 473].

Although, as briefly argued, the search for a particular development method for FLOSS is elusive, there are some indications for an answer. In particular, the so-called agile methodologies present interesting features that dialogue with FLOSS and collaborative software production in general. Agile methodologies define a set of principles and practices that deviate from traditional linear and documented methods. They emphasize incremental development, interactivity with users, and a more hands-on approach. As written in the Manifesto for Agile Software Development, "[i]ndividuals and interactions over processes and tools", "[w]orking software over comprehensive documentation", "[c]ustomer collaboration over contract negotiation" are preferred [Beck et al., 2001]. Besides customer collaboration, other points address the planners/makers duple, showing an inclination for a more direct approach in which detailed planning and document overload is not recommended. Delving deeper into this manifesto, there are twelve principles, some of them resonating clearly with FLOSS practices; for instance, the recommendation for the frequent release cycle of a piece of software, the acknowledgment of changes even in late stages, working software as a measure of achievement, the importance of team motivation, or even the value of simplicity, which in some way echoes the Unix philosophy. There are, of course, other aspects unrelated to FLOSS or that directly contradict its modes of production, such as "[t]he most efficient and effective method of conveying information to and within a development team is face-to-face conversation" [Beck et al., 2001]. Interpreted literally, this quote does not comply with the asynchrony commonly associated with distributed software development. However, there is a set of practices shared between FLOSS and the agile construct that bring both technical spheres together. Despite the various incarnations of the agile methodologies, notably SCRUM[6], extreme programming, abbreviated XP, closely reflects the FLOSS ethos. As its name suggests, XP stresses the importance of coding as the project's main force. Although the ideas proposed in XP are not new and are as old as programming [Beck, 1999], its practices embody the agile ideal, encouraging small releases, simple design, and continuous integration, among other practices. Two particular practices are of special interest, collective ownership of code, and pair programming. The former is, indeed, enforced in FLOSS using the version control systems, although, as explained before, not all users have access to all the code. The

[6]This methodology, which comes from another field, was appropriated by the software development community.

second one implies collaboration, but demands that the two programmers sit side by side to write code [Poonam and Yasser, 2018, 95]. However, with computer networking, pair programming has evolved into a distributed model; in other words, the two programmers can be in different locations [Poonam and Yasser, 2018, 95].

The initial discussion brings a more profound assessment of software and technology back to the forefront. As addressed in previous chapters, the software crisis, the rhetoric of FLOSS, and the emergence of the discipline of software engineering are built around the question of whether software can be developed as a mass-produced product. On the one hand, some stakeholders pursue a factory-like metaphor where software is *built* using measurable methodologies, applying specific tools at specific stages, and managing teams, as in a typical organization. On the other, is the idea of software as a craft, as a chaotic undertaking for which planning is futile. Clearly, this depiction is an oversimplification for the sake of argument. However, it reveals the conception of the field regarding software development and a possible spectrum where modern collaborative modes of software production move. Therefore, a software development methodology is essential, and its selection represents an underlying technological philosophy. In the case of FLOSS, it can be said that, contrary to the idea of the monolithic bazaar methodology, the distributed software production agency leads to a diversity of methods that cannot be reduced to a simple metaphor.

Nevertheless, agile methodologies and particularly XP are important, not because they represent the default methodology of FLOSS, instead, because they show how collaborative practices and a hands-on ethos have become a common element of software development. In this sense, Warsta and Abrahamsson directly assert that FLOSS (mainly open source) and agile methodologies are very close. They consider incremental processes, collaboration, and adaptation as common values [Warsta and Abrahamsson, 2003, 4].

5.6.3 Design artifacts: patterns

Software patterns came into the spotlight in the 1990s, but have been always used [Wilson and Aranda, 2011, 469], perhaps tacitly. Patterns are a problem-solving approach that offers a solution to a particular issue by providing a set of practices distilled by the community to be used once the situation is identified. In other words, there are rules for deciding whether a problem is a common problem. Similarly, there is a specific course of action or a community proven solution. Patterns do not originate in the field

of software, but from architecture; they were proposed in the seminal book "A Pattern Language. Tools, Building, Construct" [Alexander et al., 1977]. Two observations on the original idea of a pattern are of interest to the purpose of this work. These are that patterns can be modified, and patterns are languages that have the structure of a network [Alexander et al., 1977, XI, XVIII]. The first observation reveals an open ethos similar to FLOSS practices, that is, that the community can use the existing patterns to build upon new ones. The second highlights the abstract nature of the pattern exercise and its symbolic representation through language. It also highlights the network metaphor. This observation is similar to one made by Hannebauer in his doctoral thesis on patterns and FLOSS. He describes patterns as "nodes in directed graphs" [Hannebauer, 2016, 85], meaning that the structure of patterns is not linear, and their network shows a semantic complexity open to the addition of other nodes. A list of patterns, including their classification, can be obtained in [Fowler, 2003].

As an abstraction, patterns are also connected to agile methodologies, particularly to XP [Coplien and Harrison, 2004, 10], with the practice of pair programming identified as a pattern in itself. As in the case of XP, patterns and their use can be interpreted as an abstraction conceived under the idea of software development as a manufacturing process because patterns can be evaluated as tools, ready to be applied to streamline the entire process. As a matter of fact, the very idea of a software factory in which software can be assembled aided by tools, frameworks, etc. is based on patterns [Greenfield et al., 2004]. However, as this same source states, the factory simile does not imply a mechanical, factory-like process in which all creativity has been separated from the process. Instead, it is an effort in identifying practices that can be abstracted and reused, as in the case of patterns. In this sense, patterns are simply another type of nodes/artifacts in the socio-technical network of modern collaborative software production. In contrast, in the automation of the human-machine assemblage imposed by code-centric activities, such as code and other textual artifacts version management, or network-supported communications, patterns embody a more social-based abstraction. They are at the service of simplifying and rationalizing a seemingly chaotic process using common and communal knowledge, systematization, and collaborative agency. Coplien and Harrison insist, "[b]ut a deeper knowledge of patterns will make it easier for you to extend the pattern language with your own patterns, and to experience the joy that comes with the freedom of playful and insightful organizational design" [Coplien and Harrison, 2004, 14]. As in the case of XP, the collaborative features of patterns raise the question of how close they are to FLOSS. To address this topic, it is important

to consider that patterns are a product of a community and a strategy to encourage reusability [Gamma et al., 1994]. In this work, communal and distributed agency have already been postulated regarding the socio-technical network that supports FLOSS development. However, reusability and abstract artifacts depart from the common interpretation of a methodology as a previously conceived plan. They highlight experience and observation as a path to improving a complex process. In doing so, the idea of a factory is reassessed in the light of the benefits of some organization; this erodes the indebted hacker myth of software development as an exclusively chaotic profession. Specifically, Hannebauer pursues a pattern language for FLOSS development, recognizing the diversity of methodologies in the FLOSS community and identifying useful vernacular FLOSS patterns, such as "user contributions" and "exposed architecture" [Hannebauer et al., 2010, 1,2].

To summarize this section, three abstract artifacts were selected and discussed to shed light on the complex socio-technical network of collaborative software development. Like other nodes/artifacts, abstract artifacts are involved in a continuous communication process between the community (diverse and heterogeneous) and a set of automated software artifacts. Their combined agencies can produce timely works of collectively developed software in a geographically distributed fashion. However, abstract artifacts constitute a social meta-layer of consensus that, at another level, shows the collective nature of standards and agreements. This is worth noting because this type of knowledge emerges from a set of communal practices almost spontaneously.

In contrast, consider the so-called Unified Modelling Language UML, another abstraction for modeling and design whose size and complexity have made it a bloated tool that is taught mainly in systems engineering courses but whose use is not widespread in the industry [Terceiro et al., 2012, 471]. Undoubtedly, there are more abstract artifacts, but the ones selected show that collaboration tropes are required for collective and distributed agreement, and they are an alternative to the planner/maker dichotomy and the hacker rhetoric of disdain for planning. On the last point, it should be noted that these abstract artifacts do not hinder the programming and craftiness. Instead, they offer a community a structured practice that frames the freedom that is often linked to FLOSS development. Furthermore, because of the stabilization and the industry's adoption of FLOSS, as well as the presence and use of automated software tools, there is a corpus of data that can be used to infer common trends, as the renown "Empirical Software Engineering" suggests [Wilson and Aranda, 2011]. Lastly, the presence of abstract

artifacts can be useful to capture nuances within FLOSS. For example, it can be said that following the methodology; there is no difference between open source and free software. However, concerning licenses, evident differences arise [Hannebauer, 2016, 82].

5.7 Summary: the benefits of an open structure

As already suggested, the approach of this chapter builds on the historical narrative of the previous two chapters to embark on a more structural discussion that aims to clarify the complexity of software and particularly FLOSS as an embodiment of modern collaborative modes of software production. This fundamental objective focuses on the idea of the socio-technical network and the possibilities of the network topology, hence, the network metaphor. By definition, the network topology is open, and always prone to extend its borders to include more nodes; in this case, the artifacts-nodes responsible for the human-machine interaction that make software development possible. Like all metaphors, especially technical metaphors, the network mentality indebted to the emergence and expansion of computer networks, and the network society reflects how society shapes common values and practices using current referents. However, these practices are mostly part of the human condition and are explained differently depending on the metaphor of choice. Therefore, the digression of approaching collaboration from a more general perspective serves to frame software in the history of human undertakings. Although the discussion was brief, two elements were visible. The first is that collective and distributed development of software is, in fact, a form of collaboration or, more generally, cooperation. However, it does not obey the features of centrality and organization of other activities and symbols. Therefore, communication is a fundamental part of collaboration. Furthermore, the social dimension of collaboration is perceived in all software development that takes place thanks to the dynamics of the socio-technical network.

For the above reasons, the structure of the socio-technical network is presented at three levels, the circulation of textual artifacts, the composition and politics of teamwork, and the meta-level that some abstract artifacts contribute to collaborative modes of software development. First, the most important textual artifact of software is code. While this emphasis contradicts the favored definition of software that supersedes code-centric sim-

plification, it is undeniable that the level sophistication and automation of software artifacts, such as repositories and version control systems that store, manage, and distributed code, has a huge impact on the distributed agency of FLOSS development. However, a broader sense of software is recovered by considering text as the core of communication artifacts, such as IRC, wikis, forums, etc. Clearly, less governed by algorithmic tropes but with a fair level of automation, these artifacts provide for social discussion, planning, and the continuous improvement of software based on community interactions.

Consequently, one of the main characteristics of FLOSS development is the consolidation of geographically distributed work teams with various tasks and responsibilities using these communication channels (synchronous and asynchronous). It is worth noting that these geographically distributed teams are not so diverse and, contrary to the popular bazaar rhetoric, show different levels of hierarchy and rights over the source code. Finally, this heterogeneity manifests itself again in the appearance of abstract artifacts, which can be assumed as a meta-level composed of common practices distilled from collaborative community exercises. In doing so, not only do these abstracts artifacts show that FLOSS remains a complex matter, but they also offer a negotiation to plan and perform simplifications embedded in the rhetoric that surrounds this type of software. Similarly, the lack of a unified development methodology or distribution license is a genuine consequence of the richness of FLOSS development, which, in turn, ensures the potential for common agreements necessary to achieve this level of abstract artifacts. This can be interpreted as another product of the structural openness of the socio-technical network.

6 — Conclusions

6.1 Summary

The primary objective of the previous chapters was to observe the evolution of open collaboration practices in the software culture through philosophical, historical, structural, and critical lenses; this has unfolded a set of important discourses and findings. As methodologically planned, a foundation in the philosophy of technology placed various classic tropes of the evolution of technology and our assessment of its qualities into an updated software studies framework. This effort provided most of the general topics that were further developed using a dual approach consisting of a historical and structural perspective applied to the main inquiry. In doing so, other important issues arose that built a conceptual and discursive network in which the research found space to move. Thus, the narrative was drawn out through three different stages, each with a purpose and contribution to the main arguments, which are summarized below.

First, the philosophical section featured the idea of technology as a form of knowledge, challenging the traditional Western dualism where technology is placed in a secondary place, subordinated to science, which is conceived as the holder of true thought. This vindication is commonly addressed by Science and Technology studies. This dichotomy can be traced back to the episteme and techne duality, first raised in the Greek world. Its historical evolution has driven ideas, such as the emergence of technology as a byproduct of modern science in the XIX century, prompting a modern critique that has re-evaluated technology production modes and the value of manual labor, and the trial and error approach. Figures, such as Heidegger and his questioning of technique and Stiegler with his originary technicity assessment, have been key in supporting the argument presented. However, the premise of software as technology enabled the observation of software under this general STS framework. Thus, in the discussion, software inherits the character of technology but is conceived as a special technology artifact be-

cause of its linguistic substrate, flexibility and materiality. In turn, software is highlighted as a culture, a complex system of thought with a huge impact on modern society. Here, the concept of the metaphor is paramount to expand the understanding of the success of software in general[1], especially its open and collaborative modes of production.

Following the above mentioned general approach, chapters three and four addressed the historical narrative, covering the history of collaboration in software development and computer networks, respectively. This construction was based on the observation that modern modes of software production are entangled between software and computer networks; one cannot be thought of separately from the other. This position reproduced other assumptions related to the critique of dualisms made in the philosophy of technology and science and technology studies. In this sense, pair software-networks operate in a similar way to pair software-hardware, bringing an increased potential to a merely symbolic conception of software and underscoring the materiality of assemblages. The chapter dedicated to the history of software collaboration was based on the premise that FLOSS-like collaboration in software development has existed since the outset of software, and before the coining of terms such as free software or open-source software.

The discussion was divided into three stages. The first one focused on the 1950s, when the term, software, took hold, and software collaboration was not uncommon due to the scarcity of hardware and its high costs. Communities, like SHARE, and applications like PACT, belong to this period. The second described the emergence of interactive computing, which goes hand-in-hand with the appearance of one of the most recurrent themes in software collaboration, the hacker culture. Although problematic and mainly theorized later, its principles and tropes, regardless of their historical accuracy, still project considerable influence on FLOSS practices. Interactive computing was not only a catalyst for collaboration but emphasized computing and software's shift from calculation to communication. Operating systems, such as ITS or the still in use, UNIX, are products of this era. The third stage was developed based on milestones, such as the constitution of software engineering as a discipline and the disaggregation of hardware from software. Progressively, the latter acquired more relevance, driving the development of software as an industry. The commodification of software happened almost

[1]Chun observes that "[b]ased on metaphor, software has become a metaphor for the mind, for culture, for ideology, for biology, and for the economy." [Chun, 2011, 2], attesting the relevance of the metaphor concept in software culture.

simultaneously with the commercial arrival of the computer to households with the PC, once again, suggesting the software-hardware relationship. This technical landscape served as the background for the emergence of a suitably self-aware collaborative software movement, embodied in the construct of free/open-source software.

A similar narrative was built upon the history of computer networks, that is, a progressive account where hardware and software developments become entangled. The ARPANET achieved the common objective of building a computer network to share the scarce available computer resources at the end of that decade (1960). This achievement established most of the theory of networks and even anticipated, at that time, its communication affordances. The UNIX operative system reappeared, linked with the history of collaboration in the previous chapter. It also played an essential role as a test-bed for the development of network protocols in the 1970s, given its presence in several locations, the availability of its source code, and its embedded collaborative ethics. As a result, it was vital in the development of the home-brewed network, USENET, at the end of 1970, again, showcasing the articulation of software, networks, and personal computers as a commodity. Once the Internet became established as the hegemonic computer network, incorporating most of the folklore and practices already in use in USENET, the software-network-hardware entanglement was set to witness the emergence of the distributed on-line collaborative mode of software production. This scenario provided the fertile ground for the consolidation of the FLOSS troupes, methods, and engineering practices, from which, perhaps, one of the best-known metaphors is the cathedral and the bazaar. Although it shows an effort to find some principles for an innovative practice, such as networked software development, the metaphor's oversimplification of these practices was discussed from a critical perspective. All of these stories were connected through an examination of documents and statements describing collaborative aims and procedures to create a historical approach to open collaboration in the software culture.

The fifth chapter presented a structural approach to modern open and collaborative software development. The historical narrative of the creation of FLOSS and collaborative computer networks is placed in the background. Thus, the mentioned entanglement of software-networks and its collaborative substrate provides a framework in which collaborative and open modes of software production are presumably established and have a well-structured form of operation. A digression is made to frame collaboration in a more general context in an effort to describe how the socio-technical network of

artifacts of this mode of software production is executed. This movement was intended to contest dual assessments of technology using problematic oppositions. Overall, the proposal presents collaboration as a human process and briefly discusses some of its economic, motivational, and social aspects. It specifically underlines the belief that software collaboration appeared after the emergence of FLOSS. This section continues with the theoretical reference of the actor-network theory, considering the socio-technical network of software production as a flexible set of nodes/actants in constant network communication. With this in mind, it describes some nodes/artifacts. The first is, perhaps, the node/artifact that stands as the backbone of modern collaboration, the version control system, which allows code revisions, forks, and distributed collaboration to be achieved seamlessly. However, this code-centric view of software is not without tension, as it highlights the relevance of code as a fundamental artifact, in fact, a distributed code object. At the same time, it calls attention to the commonly oversimplified miss-conception of software as code. Communicative artifacts are thus addressed to counterbalance this issue. Like code, communicative artifacts are based on text. However, communicative artifacts offer the parties involved in a project a set of text-based media to coordinate and discuss development-related activities. It is not an algorithmic text machine intended to be executed by a piece of hardware in the traditional way. Thus, the text of communicative artifacts follows the logic of human languages rather than programming languages and therefore consists of mechanisms, such as mailing lists, wikis, IRC channels, and issue trackers; there are others, but these were selected for the argument. Finally, two additional groups of complex artifacts are considered, which are not as software-based and depend more on social consensus and negotiations than on automated human-machine interactions. The first group deals with motivation. In FLOSS, the common rhetoric of motivation points to volunteer work as the main force in collaborative software development; however, this aspect is far more complex. Some common roles of the community are described to round out this issue. The second group included so-called meta-artifacts. Licenses, methodologies, and design artifacts such as patterns are included in this category. More specifically, these artifacts are not direct actants of the collaborative socio-technical network of software production; instead, they are considered useful abstractions built upon agreements (licenses) or praxis (methodologies and design artifacts), and their role among the community. This closes the structural section of the work; but, ultimately, leaves some questions to be addressed later. As such, this conclusion section not only includes this summary; it also develops some critical aspects, presents findings, and some topics that could be considered for future research.

6.2 Critical discussions

6.2.1 Recurring themes

Several common tropes have appeared throughout this work to support a particular argument or as a result of a certain train of thought. Generally speaking, these topics resonate with a broader theoretical framework and are therefore compatible with the current state of the art in fields such as software studies, software engineering media archeology, and science and technology studies to name the more relevant. These topics form a fabric of intertwined constructs that constitute a platform where the mentioned two-headed approaches of historical and structural perspectives on software culture and collaborative modes of software production are developed. References to these themes are repeated throughout the text and function as an index for the more general adverts of the current discussion in the media landscape that are too extensive to be addressed in detail. Although they have been briefly discussed in several places, a brief list is provided below with their respective interpretation and relationships with each other from the perspective of this work.

- *Controversial dichotomies*: This is a common discussion inherited from Latour's works (particularly in [Latour, 1993]), and science and technology studies. Specifically, modernity is described as being built on opposing constructs, with the subject-object and culture-nature being the most renewed, a tradition that is considered inconvenient because of its inaccuracy with real issues. The software culture, in turn, has its own set of dualisms, like virtual-reality, material-immaterial, software-hardware, physical-mental work, and making-planning, among others. These dualities are equally troubling, not only because they have some impractical consequences and inadequate work hypothesis when approaching software, but also because they underscore the idealistic substrate behind the subordination of technology to science or manual to mental work. The importance of this assessment is that hybrid constructs surface as potential answers to the posed dilemmas, underscoring the complexity of some assemblages, like software.

- *Relationship between science and technology*: This was mainly discussed in Chapter Two. Following the construct established by the classic Greeks, Western tradition did not attribute knowledge to technique. This stance evolved into a duality between science and technology, where the latter was depicted as a byproduct of the former.

Although modern technology would be unthinkable without science, such an oversimplification leads to the absurd conclusions that technology appeared after the emergence of modern science after the XVII century or after the industrial revolution. Moreover, it detaches technology from the very human condition that characterizes itself as a species that produces technology. The second position is related to the concept of originary technicity, which is used to provide an ontological dimension to the close entanglement between humans and technology. From the perspective of software, as this is part of technology, it features similar constructs; for instance, the assessment of software as a science, with calculation and defined methods (computer science), or its conception as a craft, a chaotic hands-on approach (hacker culture rhetoric) with software engineering in the middle. Although software cannot be ascribed to any of these groups exclusively, its complexity is manifested in the tension between these readings.

- *The materiality of software*: This is one of the most common topics of software studies, the constant affirmation that software is material. The standard rhetoric of software as immaterial at the outset of the discipline has been regularly considered imprecise and misleading. Although it can be seen as a relic of the techno-utopic discourse that prevailed even into the 1990s, it is still used in some common buzzwords, such as immaterial labor, which is connected to software intermediation. The materiality of software has been argued from several fronts, from the electrical pulses at the base of stack abstractions, the highly visible outcomes of software operations, and the materiality of written discourse and agency of computer orders, to the materiality of infrastructure and labor, among others. Although they are all relevant in discussing how open and collaborative modes of production affect software development, the latter is of utmost importance because infrastructure and labor are very present in the actor-network theory, which, applied to software, sees its process as an articulation of actants with alternating communication between human and automated processes on several platforms, mainly computer networks, such as the Internet.

- *The Actor-Network Theory*: The ANT (acronym) is sometimes criticized for not being a complete theory in the traditional sense. Its basic components play an essential role in understanding the complexities of the software assemblage, involving a group of people, underlying infrastructures, technical and social artifacts, and procedures that are not

fixed and vary in various ways. In this sense, the concepts of networks and actants are heavily used to explore different components of open and collaborative modes of software production. Networks obviously refer to today's indispensable computer network infrastructure that is required for software development. However, it even refers to the abstraction of the network as a changing system, as well as a metaphor that is reified through open collaboration in the development of computer networks and the software-hardware-networks triad. The idea of actant also becomes useful to describe the different nodes of the mentioned network, which can be humans with different roles in software development or specific software pieces that support software development by providing communication or automating different tasks. The actor-network theory is part of science and technology studies and, therefore, can be applied as a technology that is considered to be as complex as software.

- *Metaphor*: As stated at the end of Chapter Two, the software culture is, in part, indebted to its ability to generate metaphors. This appreciation, elaborated here more clearly, has two levels. First, the software culture has developed metaphors to express its own constructs; this has been indispensable at different stages of software history when virtually no precedent existed for a particular concept, and a figure of comparison was required. Take, for example, metaphors like *Desktop* or *Client/Server* that represent specific technologies and how they can be better understood. The second level, however, is perhaps the most important and constitutes one of the main features of a culture. In this case, the software culture goes beyond needing to borrow metaphors from other fields and begins to provide its metaphors to other realms; this is the basis of one of the main arguments of this work. The software culture, and particularly its collaborative and open forms of software development, have impacted the so-called network society so profoundly that it has started to be seen as a source of concepts, tools, and models to achieve objectives in entirely different fields.

- *Performative text*: Although this specific term was not used, it was tacitly presented throughout the argumentation, pointing to the multiple nuances of written text as a performative inscription. It clearly draws attention to the dual condition of text in software, and how programming languages can be studied under the same linguistic parameters as natural languages. The dual condition of text refers to how, on the one hand, it is intended to be written and read by both

humans and machines and, on the other, to be executed by machines. The nature of code as a communication means between humans, and between human and machines, produces several levels of agency among the socio-technical network of software development. As stated in the fifth chapter, text prevails in code-centric artifacts, like version control systems. However, it is also at the core of distributed human to human communication using natural language; this can be attested in communication artifacts, such as wikis and mailing lists. An innovative software studies discipline, such as code studies, commonly address these programming language considerations and text, in general, as an abstract. However, understanding software requires material artifacts.

- *The hacker culture*: This socio-technical construct informs most of the discussions on topics, such as collaboration and openness in software culture, and it is strongly tied to the FLOSS history and narrative. The hacker culture encompasses a set of values and ethical principles, which are set out in Stephen Levy's well-known account. It is a socio-technical culture with a strong influence on several technical areas, not just software. This last observation, once again, underscores the software culture's metaphorical quality. The hacker culture develops several rationales at the base of possibly the most acknowledged narrative of collaboration in software. Constructs, such as the hands-on approach, relevance of community, and dualism between planners vs. makers, underline a form of software production, based on virtuosity and laissez-faire management that points to the romantic gesture of associating craftsmanship with open and horizontal collaboration in software. Although the hacker culture was theorized late, it is still very influential despite its inconsistencies.

6.2.2 Views and findings

As is usually the case with endeavors such as this work, a research investigation articulated a corpus of previous work with its own interpretation to provide an innovative narrative on a particular topic to shed light on a relevant issue. To this end, this work analyzed the open and collaborative modes of production in the software culture through the mentioned historical and structural approaches. Although most of the references and tropes are already known, the value of this effort is the new connections made and the organization provided, which emphasizes some points over others to provide arguments that support the selected way of proceeding. This articulation of previous works is built upon to produce new explanations that, in the aca-

demic tradition, contribute to existing stances drawn from past experiences and new considerations. The abbreviated list that follows describes some of these new developments. Some have been addressed cursorily in previous works, limited by time and space, others are new developments that to the author's knowledge, have not been specifically mentioned in those works. The significance of these views is that they are transversal to the entire work and interrelated, producing an argumentative network that fits the research objectives.

- *Algorithmic agency*: This is the simplest kind of agency in this work's interpretation of software. From an early stage, the code's dual condition (read and executed) was considered when no other artifacts (documentation, for example) constituted the definition of software. In this design, a unit of software modeled an algorithm, which included a course of actions framed by a set of rules. From a historical perspective, the algorithmic agency described a computational model, embodied in the batch processing paradigm. In other words, the programs ran without intervention after the inputs were entered, with the results being the most important evidence of the process. In this sense, the algorithmic agency reflects a cybernetic reading of software as a black box with inputs and outputs. This stage marks the departure of software from scientific endeavors purposed solely to conduct and automate calculations carried out, until then, by human computers. In this sense, the topology of the algorithmic agency is linear.

- *Interactive/modular agency*: This stage, which coincides with the emergence of interactive computing in the 1960s, builds upon the algorithmic agency. As in the previous case of agency, these observations mark a particular computing paradigm (and its change) as a framework; in this case, the shift from batch processing to interactive processing. From the perspective of the agency, interactivity represents a clearer software-human assemblage as human intervention is not expected only at the beginning or end of an algorithmic process. It can occur during some of the stages or at any point in a computer program, as in the command line. Similarly, from a sociohistorical stance, computing (thus, software) ceased to be a task charged to a limited priesthood with access to isolated computers, it began to spread to a broader public. Therefore, this narrative is tied to time-sharing systems, with the operative system as the epitome of interactive software. Structurally, interactivity required more complex artifacts; this resulted in the emergence of modules, structured programming, and the discipline of software

engineering, in general. The software used for scientific projects originated the line of software evolution embodied by the hacker culture. Moving past the possible criticism, the form of the hacker shaped the collaboration narrative and underscored the relevance of communication in computing. The topology of the interactive agency became the flowchart or automaton, which required the constant input by a user to give the human-machine entanglement execution capability.

- *Distributed agency*: This is the last layer of the agency stack. As its name suggests, it is based on the existence of computer networks as infrastructure. As the network society construct attests, computer networks represent the complete transition of computers from calculation to communication machines, placing the stress of the collective agency of hybrid artifacts required to accomplish the open collaboration of software modern modes of production on human-machine interactions. As in the former level, interactivity is paramount. However, instead of the time-sharing model that favors a type of client/server centralized scheme, computer networks encourage a set of connected devices. This new approach enabled geographically distributed and asynchronous communication. Thus, it provided a human-machine agency that was not explicitly located in one place but dispersed in a layout of different actants or node/artifacts whose articulation granted the conditions of open collaboration. The conclusion is that the agency of the entire assemblage cannot be attributed to a particular set of social or technical artifacts, nor the mere sum of the parts but the continuous flux of mainly textual artifacts (code and written communication) and different negotiations between humans, machines, and human-machines with some of these negotiations being automated by the software. For this reason, the topology of this type of agency is obviously rhizomatic, like a network[2].

- *The collective*: This construct is a compound of the community and a set of technical artifacts. As a hybrid concept, it responds to the nature of an actant. It represents non-human or human entities with their own agency. Throughout this work, the actant has been interpreted as singular; the collective, however, refers to a group of actants. The human members of this compound are specifically referred to as the community, a term, which is widely used in the FLOSS field. As described in Chapter Five, it includes a group of people with differ-

[2]Distributed agency is an own elaboration based on the concept of the same name found, along "social operative system" , on [Krieger and Belliger, 2014].

ent roles, such as programmers, documenters, issue trackers, etc., that collaborate openly over a computer network infrastructure. The community is a fundamental part of the FLOSS and hacker narratives, in which, from an idealistic perspective, voluntary work and altruism are stated as its core values. The other part of the compound refers to technical artifacts. Like the community, the technical artifacts are diverse and range from fully-automated code support devices to enablers of human discussions. The main feature of the collective is that it performs the previous layers of agency distributed in time and space through a socio-technical network.

- *The socio-technical network*: This was developed on the construct of a socio-technical system used in science and technology studies that considers technology a part of the social fabric. Although it may seem like a simple renaming, in reality, this distinction between system and network reflects an important stance that signals again to a historical and a structural perspective. On the one hand, the concept of networks appeared, technically and metaphorically speaking, after the system. With this shift, a change in a conception that favored control led to a more communication-based construct. On the other, although systems are also open and have parts, they tend to be seen as more hierarchical and organized, while a network represents a chaotic but functional quality of continuous increments. This can be summarized by pointing out that every network is a system, but the implication in the other direction does not always apply. Therefore, the network topology is better adapted to the complex socio-technical interactions behind every technology, let alone the software that has proven to be so influential, social, and human.

Although the above views have been described in isolation, with some references to others, they operate interconnectedly. Even though the three layers of agency appear successively in the evolution of software and computing, they cannot be assessed in a linearly. To somewhat paraphrase McLuhan, none of these constructs displaced or surpassed the other. Instead, they articulated together seamlessly in open and collaborative modes of software production. Consider the automated artifacts that require virtually no human involvement, such as nightly code builds, interactive communications mediated by software, or distributed commits to a codebase; they are all afforded by the different artifacts dedicated to collaboration. These three agencies are performed by the collective, that is, an assemblage of hardware, network devices, pieces of software, and a diverse community, in the time and

space provided by the socio-technical network. The description of this model has been tacitly made in various sections, and, although it can be considered incomplete, as is usually the case, it seeks to allow more flexibility in the representation of the complexity of modern software production and on how an assemblage materialize open collaboration metaphors.

6.2.3 Problematic aspects

This brief section highlights some possible criticisms and observations. The former most likely involves how the proposed narrative is developed and the topics that were excluded from the main body. This exercise is intended to clarify the scope of this research and establish possible aspects to explore in future works rather than to defend a particular position. To approach the latter intention, the following section will address a specific field of interest to encourage the study of the hybridization of the software culture and open metaphors from a global south perspective.

For starters, we can look to the very purpose of this work. As stated in the introduction, part of the study's objective was to address open collaboration in software production, separating it from the FLOSS movement, which can be seen as a late example of this collaboration. Although the relevance of FLOSS cannot be overlooked, and it is part of the presented narrative, a question arises. What can the contribution be if there are several works on collaboration in software and FLOSS? In addition to the distinction made previously (FLOSS is a particular case), further clarification is due. From the late '90s to the first decade of this century, academic undertakings involving FLOSS were hyped. Many of the threads of software discourse of the time are now being contested. For instance, the persistence in the idea that immaterial and voluntary work are the main drivers of FLOSS development. There have also been parallel narratives from activism and journalism that present FLOSS in opposition to commercial software. Thus, one of the contributions of this work is an updating of the appraisal of open collaboration in general and FLOSS in particular at a time when these modes of production are no longer a novelty and are embedded in society. Some of the better known academic efforts on FLOSS from that time are [Grassmuck, 2004], [Kelty, 2008], [Berry, 2008], [Taubert, 2006] and [Chopra and Dexter, 2008], to name a few. Two key journalist accounts that have been extensively quoted can be found in the celebrated Levi[Levy, 1994] and Moody [Moody, 2002].

The second point to address is how the provided narrative supports the objectives. Part of the approach adopted is carried out by following a histor-

ical path, a decision that usually involves several complexities. It is prudent to highlight here that this is not a comprehensive history of computing or software. The elements are arranged to provide a progressive assessment of the evolution of open collaboration in the history of software and computer networks. As is often the case with written history, perceived linearity obfuscates the rich diversity of facts, evolutionary paths, and traditions in favor of a distinct hegemonic viewpoint. Consider, for instance, the account presented on computer networks. It corresponds to a ubiquitous course that traces the modern Internet back to Arpanet, and, in this case, includes UNIX and its development in USENET as a small counterbalance. This choice is based on two observations, that Arpanet displayed collaborative features and was the first successful computer network, and that modern modes of collaborative and open software production depend on the Internet. However, this does not discount the existence of other former or current networks (e.g., Cyclades in France, NPL in England, or the failed OGAS in the then Soviet Union). Instead, it stresses the fact that the Internet was the project that took over the computer networks imaginary and absorbed other networks, creating the infrastructure used by the socio-technical network of collaboration.

Another observation of the historical account is the tendency towards some heroic personal sagas, mostly of men, who are, almost single-handedly, responsible for a disruptive change; the hacker culture folklore is full of these cases[3]. Although this remains a possibility, in general, it contradicts the social and collective nature of technology development. Some of these well-known stories have been included because of their value to the community. Related to the last claim, a methodological tool must be acknowledged. Various technical documents and papers have been consulted and quoted to distance ourselves from the described features of the historical accounts and provide a more precise narrative. This allowed a shift of focus from the healthy thread of the FLOSS community's oral tradition to written evidence dating back to the 1950s that supports the premise that open collaboration in software predates FLOSS. This effort achieved a more general framework of software studies.

Lastly, there is a very important set of observations that are at the root of any future work. These considerations concern the tension between the hacker culture narrative and its aspiration of universality, as well as its lack

[3]On this point, Ensmenger observes how historians have favored big names, in the industry or in the hacker culture, overlooking "the silent majority of computer specialists"[Ensmenger, 2010, 3].

of diversity. In other words, the hacker culture is at the core of the metaphor of openness and collaboration that has been so successfully adapted to other fields, and that has hybridized with other cultures. However, as partially explained in Chapter Five, the hacker culture refers to a white, male, Western, euro-centric vision that does not contemplate other ways of approaching technology, particularly software. The statistics of the development of FLOSS confirm this appreciation notoriously; they show that most developers are males in the United States and Western Europe. Thus, the historical account refers to American figures almost exclusively, attesting the incontestable fact that American companies dominate most of the software industry. Because the commercial/non-commercial software opposition is an oversimplification, this dominance is also transferred to FLOSS. Here, the cathedral and bazaar explanation is noteworthy because it reproduces the dichotomies already disputed in Chapter Two and does not recognize that collaboration can also occur in closed projects. A partial conclusion is that, like technology, open collaboration forms in software and network society, in general, have a boundless potential that has not yet been achieved, despite the success of several free and open projects and their increasing significance, not only in software culture but also in culture in general.

6.3 Future work: metaphors from the south

[4] The previous observations on some problematic FLOSS tenets and the open collaboration narratives create a platform to develop further research that addresses these issues. The position is clear; this work attempts, from a historical and structural perspective, to convey the evolution and success of open collaboration practices in the software culture, which, in turn, is partly responsible for the emergence of the network society. As a culture, software has provided not only a sound technical solution to different modern problems but also a model of production that can be extrapolated. By definition, culture provides a set of metaphors; in this case, they are represented by the idea of open collaboration by networks, which can be interpreted from different perspectives. Various groups have adopted the rhetoric of openness and collaboration provided by software technologies, from big companies to activist collectives. This statement shows the fallacy embedded in the commercial/non-commercial software opposition in which the open and collaborative qualities are considered an exclusive asset of non-commercial

[4]Besides the software metaphors discussed in this work, this title refers to other sociotechnic metaphors coming from the global south and their role in expanding our technical understanding

software. To illustrate the tension within this spectrum, consider that, on the one hand, this collaboration of open imperatives has been adopted mainly by software companies, not only in their FLOSS-compatible infrastructure but also in their public relations discourse. On the other hand, different activist groups frequently refer to FLOSS ethics as a source of inspiration. Both sides are problematic. Collaboration and openness can obscure free labor exploitation processes or advertising objectives that show the success of the label, or a non-critical oversimplification of narratives such as the hacker culture, which has become exceptionally popular among activists. As argued, this is not only a validation of the relevance of the software culture and its metaphors, of their flexibility and adaptability, but also of the contradictions they entail. The most perilous contradiction is that although the software culture's open and collaborative metaphors with its characteristics of diversity and social flatness are seemingly very persuading, the universality of corporate discourses and their agendas, as well as the so-called *Californian ideology*[Barbrook and Cameron, 1996], jeopardize the alleged heterogeneity of metaphors transferred from the software to the network society. The conclusion of this brief introduction is twofold. Although the open and collaborative metaphor represents the software culture's ability to go beyond the merely instrumental perception of technology, a growing homogeneity threatens the supposed diversity of constructions, such as FLOSS. It is precisely the global scope of information technology, which shows software's complexity and the inaccuracy of binary constructions concerning socio-technical networks, that offers the possibility to hybridize the open metaphor with local cultures different to the Western. It provides an alternative to the dominant techno-determinist corporative narrative. An example of this possibility is embodied by the actions that try to position the southern (global south) perspective.

This "global south" construct is not new. It is relevant in several aspects of modern life, mainly political and socio-economic issues, in which the gap between developed and other countries is addressed. This term replaces previous ones, such as "third world" or "non-developed countries". Despite not having a fixed definition, it is used increasingly in academics, including science, hence, in technology. Regarding science, it exposes the hegemonic place that science gives Western thought, creating a conflict in which rationalism is considered true and other forms of thought are disregarded. This phenomenon is considered a type of "epistemic violence". Therefore, there have been advocacy efforts towards a more holistic and diverse assessment of science; one such case is the so-called epistemologies of the South [de Sousa Santos, 2009]. A similar observation can be made on technology as a dominated

system of thought with a misleading techno-narrative. However, this does not imply –as argued in previous chapters– a view of technology subordinated to science that inherits these problems. Instead, it indicates that it is precisely this widely accepted view of the science-technology relationship that global capitalism has exploited to show that technological change develops from big investments in science, dismissing non-western, non-science-informed, or indigenous developments. From that, it follows that technology is interpreted from the top down. Hierarchies in the Global South and population, in general, perceive technology, especially information technologies, as a solution transferred from the west, "from a globally networked to a remotely disconnected" [Chan, 2014, 184]. Governments in particular generally adhere to this rhetoric of techno-determinism and techno-solutionism where technology is thrown at a problem to solve it [Chan, 2014]. These positions are highly problematic, besides the obvious, because they assume an universality of technological discourse that does not engage with local and vernacular needs and practices [Fressoli et al., 2014, 47].

Similarly this hegemonic posture oversimplify the negotiation process known as *"Translation"* (Latour), ignoring the contingent and changing nature of techno-scientific objects [Chan, 2014, 186] or they plainly assume the deployment of noncritical technology as the best scenario without even asking the population whether they want to participate or not [Osio, 2016]. From the perspective of open collaboration, or, more precisely, from Open Source, some readings of how its stories are positioned with respect to the Global South can be briefly outlined. First, the mere construct of FLOSS so closely linked to the open collaboration reading of software is unstable. As described in previous chapters, FLOSS is an entanglement between free software –a term coined in the 1980s– and Open Source, a more business-friendly term that appeared in the late 1990s. As stated in Baybrooke and Jordan [Baybrooke and Jordan, 2017], open-source is a stripped-down version of free software where the community aspect of collaborative software development was understated in favor of technical superiority, a move to attract corporate support. In doing so, open-source created one of the modern techno-myths, which are characterized by "their technological determinism, their assumption of the values of neoliberal capitalism and the various way they may render Global South and non-Western perspectives invisible" [Baybrooke and Jordan, 2017, 26].

Moreover, and as a manifestation that the open metaphor from the software culture can carry such unique concerns and narratives, consider the so-called maker movement, apparently, a culture in itself. On the one hand, sharing designs, ideas, code, and instructions to produce objects are some

tenets of open software collaboration shared by the maker movement/culture [Mestres and Hinojos, 2017, 9]. These types of descriptions may seem like a reification of the abstractions of software that are part of the conventional idea of the maker movement. However, they reflect the same previously mentioned context with a hegemonic narrative that stresses heroic accounts (manufacture is the new industrial revolution) and commercially co-opted grassroots initiatives. The result is the same. The maker techno-myth serves the neoliberal interest and legitimates itself through originality, disregarding other non-Western cultures, such as the *Jugaad* in India or the *Shanzhai* in China, with similar qualities, but that predate the maker movement and do not share the same corporate vision [Baybrooke and Jordan, 2017, 29].

This overview provides the framework of the Global South to propose a promising research question that may lead to future research efforts. Following the idea of situated knowledge, rather than the entire Global South construct, the issue would be focused on Latin America. A fitting question could be, how did open, collaborative practices of software culture hybridize with other techno-cultural traditions in Latin America? In the development of this work, questions on where to conduct further explorations appeared. As Valencia observed, there are two paths to challenge the hegemonic and colonial impositions of information technologies. One is *"la subversión de la lógica moderno/colonial del código de la tecnología"* (the subversion of the modern/colonial logic of the code of technology). The other is "la producción de tecnologías con otras lógicas" (producing technologies based on other logic)[Valencia, 2017, 70].

It should be noted that the first alternative resonates with the common extrapolation of hacking, in this case, hacking the entire hegemonic system. The second is represented in projects on the scale of the Free-Libre, Open Knowledge (FLOK) Society. This project was carried out some years ago in Ecuador. It attempted to model various economic, social, and political aspects after a combination of software infused with open metaphors and local constructs, such as those of *el buen vivir* and *el buen conocer* (good living and good knowledge)[5]. An approximate classification of the various Latin American manifestations of these hybrids can be provided at first glance. First, there is the techno-romantic group where open collaboration in software and FLOSS narratives are identified with ancestral ways of managing

[5]As an example of this hybridization, the FLOK Society's motto is *"Hacia una pachamama del conocimiento común y abierto"* (towards a pachamama of open and common knowledge). Pachamama refers to the indigenous Andean concept of "mother earth".

common goods and land, like in the *Ayni* (Bolivia), the *Minga* in Ecuador and Colombia, or the *Tequio* in Mexico [Medina Cardona, 2014]. The other group represents artistic collectives and projects that have adopted open technologies and hacker ethics, despite the contradictions concerning diversity, as a form of praxis in which the focus is on sharing, process, communal authoring, and low technologies [Medina Cardona, 2016]. The third group corresponds to practices such as "technological disobedience" [Mackie, 2018], where technologies are intervened, adapted, and even pirated to solve local problems and issues on scarcity. Although this rough grouping may be considered arbitrary, the value of this preliminary exercise is to portray the richness and diversity of how the software culture and open metaphors have been taken up in the region.

In conclusion, it can be said that this future pursuit must follow two premises. It must be clear that more than contesting Western narratives and refuting scientific rationalism and hegemonic techno-narratives, the idea is to critically approach them to rework their pitfalls and obtain hybrid improvements. The second premise is that these forms of hybridization are not only an example of technological development but also a demonstration of the flexibility of the software culture's open and collaborative metaphors in the global network society. These hybrid forms resonate with technology understood as a poiesis, a poetic and creative mode that goes back to Heidegger's aspiration of an alternative to scientific knowledge through this creation, which, from the perspective of this work, must include the creation of technologies through open collaboration, but mostly the creation of new metaphors that can herald future worlds. This objective can be summarized in the following quote, originally in Spanish, which refers to the philosophy of software, *"el énfasis sobre lo creativo, sin embargo, implica que la defensa de la tradición no puede consistir en el rechazo de lo tecnológico, de lo nuevo y de lo global, sino en un diálogo honesto que desemboca, probablemente, en la cultura del mestizaje y del híbrido"*, that translates as the emphasis on the creative, however, implies that the defense of tradition cannot consist of the rejection of the technological, the new, and the global, but in an honest dialogue that probably leads to the culture of *mestizaje*[6] and the hybrid [Osio, 2016].That is, the technologies that lie ahead should be open, hybrid and diverse.

[6] "Cross-breeding", "blending of cultures".

Bibliography

[Aguiar and David, 2005] Aguiar, A. and David, G. (2005). Wikiwiki weaving heterogeneous software artifacts. In *Proceedings of the 2005 International Symposium on Wikis*, WikiSym '05, pages 67–74, New York, NY, USA. ACM.

[Aiken, 1964] Aiken, H. (1964). Proposed automatic calculating machine. *IEEE Spectrum*, pages 62–69. Written in 1937.

[Akera, 2001] Akera, A. (2001). Voluntarism and the fruits of collaboration: The ibm user group share. *Technology and Culture*, 42(4):710–736.

[Alexander et al., 1977] Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language. Tools, building, construction.* Oxford University Press.

[Alkadhi et al., 2018] Alkadhi, R., Nonnenmacher, M., Guzman, E., and Bruegge, B. (2018). How do developers discuss rationale? In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 357–369.

[Armand, 2006] Armand, L. (2006). Technics & humanism. In *Technicity*. Literaria Pragensia.

[Armer, 1980] Armer, P. (1980). Share - a eulogy to cooperative effort. *Annals of the History of Computing*, 2:122–129.

[Arns, 2002] Arns, I. (2002). *Netzculturen*. Europäische Verlagsanstalt.

[Axelrod, 2006] Axelrod, R. (2006). *The evolution of cooperation - revised edition*. Basic Books.

[Bacon, 2017] Bacon, J. (2017). The decline of gpl? Accessed 10.07.2019 `https://opensource.com/article/17/2/decline-gpl`.

[Baker, 1956] Baker, C. L. (1956). The pact i coding system for the ibm type 701. *J. ACM*, 3(4):272–278.

[Balter, 2015] Balter, B. (2015). Open source license usage on github.com. Accessed 10.07.2015 `https://github.blog/2015-03-09-open-source-license-usage-on-github-com/`.

[Barbrook and Cameron, 1996] Barbrook, R. and Cameron, A. (1996). The californian ideology. *Science as Culture*, 6(1):44–72.

[Bauwens, 2014] Bauwens, M. (2014). Evolving towards a partner state in an ethical economy. *Acoustic.Space. Peer-reviewed Journal for Transdisciplinary Research on Art, Science, Technology and Society*, 12.

[Baybrooke and Jordan, 2017] Baybrooke, K. and Jordan, T. (2017). Genealogy, culture and technomyth. decolonizing wstern information technologies, from open source to the maker movement. *Digital Culture & Society*.

[Beck, 1999] Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, us ed edition.

[Beck et al., 2001] Beck, K., Grenning, J., Martin, R. C., Beedle, M., Highsmith, J., Mellor, S., van Bennekum, A., Hunt, A., Schwaber, K., Cockburn, A., Jeffries, R., Sutherland, J., Cunningham, W., Kern, J., Thomas, D., Fowler, M., and Marick, B. (2001). Manifesto for agile software development. Accessed 10.07.2019 `http://agilemanifesto.org/`.

[Bemer, 1982] Bemer, R. W. (1982). Computing prior to fortran. In *Proceedings of the June 7-10, 1982, National Computer Conference*, AFIPS '82, pages 811–816, New York, NY, USA. ACM.

[Berry, 2008] Berry, D. M. (2008). *Copy, Rip, Burn. The Politics of Copyleft and Open Source Software*. Pluto Press.

[Berry, 2011a] Berry, D. M. (2011a). The computational turn: Thinking about the digital humanities. *Culture Machine*, 12.

[Berry, 2011b] Berry, D. M. (2011b). *The Philosophy of Software. Code and Mediation in the Digital Age*. Palgrave Macmillan.

[Bezroukov, 1999] Bezroukov, N. (1999). A second look at the cathedral and the bazaar. *First Monday*, 4(12). Accessed 15.05.2018.

[Boehm and Steel, 1959] Boehm, E. M. and Steel, Jr., T. B. (1959). The share 709 system: Machine implementation of symbolic programming. *J. ACM*, 6(2):134–140.

[Bolt and Inc., 1969] Bolt, B. and Inc., N. (1969). Report no. 1763. initial design for interface message processors for the arpa computer network. Technical report, Bolt, Beranek and Newman Inc.

[Bolt and Inc., 1976] Bolt, B. and Inc., N. (1976). Report no. 1822. interface message processor. specifications for the interconnection of a host and an imp. Technical report, Bolt, Beranek and Newman Inc. (January 1976 Revision).

[Bolter and Grusin, 2000] Bolter, J. D. and Grusin, R. (2000). *Remediation. Understanding New Media.* MIT.

[Boomen, 2014] Boomen, M. V. D. (2014). *Transcoding the Digital. How Metaphors Matter in New Media.* Institute of Network Cultures.

[Bork, 2008] Bork, J. R. (2008). The free, open source option as ethic. *e-mentor*, 27(5):9.

[Boulton, 2014] Boulton, J. (2014). *100 Ideas that changed the Web.*

[Bradley, 2006] Bradley, A. (2006). Originary technicity: Technology & anthropology. In *Technicity.* Literaria Pragensia.

[Bradley and Armand, 2006] Bradley, A. and Armand, L. (2006). Introduction. In *Technicity.* Literaria Pragensia.

[Brasseur, 2016] Brasseur, V. (2016). Getting started with irc. Accessed 15.03.2019 `https://opensource.com/article/16/6/getting-started-irc`.

[Brate, 2002] Brate, A. (2002). *Technomanifestos.* Texere.

[Brooks, 1995] Brooks, F. P. (1995). *Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley Professional, 2 edition.

[Budapest Open Access Initiative, 2002] Budapest Open Access Initiative (2002). Budapest open access initiative. Accessed 16.08.2016 `https://www.budapestopenaccessinitiative.org/read`.

[Bush, 1945] Bush, V. (1945). As we may think. *The Atlantic.*

[Callon, 1984] Callon, M. (1984). Some elements of a sociology of translation: Domestication of the scallops and the fishermen of st brieuc bay. *The Sociological Review*, 32(1_suppl):196–233.

[Campbell-Kelly, 2004] Campbell-Kelly, M. (2004). *From Airline Reservations to Sonic the Hedgehog. A History of the Software Industry.* MIT Press.

[Campbell-Kelly and Aspray, 1996] Campbell-Kelly, M. and Aspray, W. (1996). *Computer: A History of the Information Machine.* Basic Books.

[Campbell-Kelly and Garcia-Swartz, 2009] Campbell-Kelly, M. and Garcia-Swartz, D. D. (2009). Pragmatism, not ideology: Historical perspectives on ibm?s adoption of open-source software. *Information Economics and Policy*, 21(3):229 – 244.

[Campbell-Kelly and Garcia-Swartz, 2015] Campbell-Kelly, M. and Garcia-Swartz, D. D. (2015). *From Mainframes to Smartphones. A History of the International Computer Industry.* Harvard University Press.

[Castells, 1996] Castells, M. (1996). *The rise of the network society.* Wiley-Blackwell.

[Cerf and Kahn, 1974] Cerf, V. G. and Kahn, R. E. (1974). A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22:637–648.

[Ceruzzi, 1997] Ceruzzi, P. (1997). Crossing the divide: Architectural issues and the emergence of the stored program computer. *IEEE Annals of the History of Computing*, 19(1):5–12.

[Ceruzzi, 2003] Ceruzzi, P. E. (2003). *A History of Modern Computing (History of Computing).* MIT Press.

[Ceruzzi, 2012] Ceruzzi, P. E. (2012). *Computing. A concise history.* The MIT Press.

[Chan, 2014] Chan, A. S. (2014). *Balancing Design: OLPC Engineers and ICT Translations at the Periphery*, pages 181–205. MIT Press.

[Chesson, 1975] Chesson, G. L. (1975). The network unix system. *SIGOPS Oper. Syst. Rev.*, 9(5):60–66.

[Chiou et al., 2001] Chiou, S., Music, C., Sprague, K., and Wahba, R. (2001). A marriage of convenience: The founding of the mit artificial intelligence laboratory. PDF. accessed 14.12.2016.

[Chopra and Dexter, 2008] Chopra, S. and Dexter, S. D. (2008). *Decoding Liberation. The Promise of Free and Open Source Software*. Routledge.

[Chun, 2005] Chun, W. H. K. (2005). On software, or the persistence of visual knowledge. *Grey Room*, (18):26–51.

[Chun, 2011] Chun, W. H. K. (2011). *Programmed Visions: software and memory*. MIT Press.

[Coelho and Valente, 2017] Coelho, J. and Valente, M. T. (2017). Why modern open source projects fail. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[Cohen, 1999] Cohen, I. B. (1999). *Howard Aiken: portrait of a computer pioneer*. MIT Press.

[Coleman, 2009] Coleman, G. (2009). Code is speech: Legal tinkering, expertise, and protest among free and open source software developers. *Cultural Anthropology*, 24(3):420–454. cited By (since 1996) 0.

[Coleman, 2010] Coleman, G. (2010). The hacker conference: A ritual condensation and celebration of a lifeworld. *Anthropological Quarterly*, 83(1):47–72. cited By (since 1996) 0.

[Coleman, 2012] Coleman, G. (2012). *Coding Freedom: The Ethics and Aesthetics of Hacking*. Princeton University Press.

[Collyer and Spencer, 1987] Collyer, G. and Spencer, H. (1987). News need not be slow. Technical report, Department of Statistics and Zoology Computer Systems - University of Toronto.

[Copeland, 2004] Copeland, B. J. (2004). Computation. In Floridi, L., editor, *Philosophy of Computing and Information*. Blackwell Publishing.

[Coplien and Harrison, 2004] Coplien, J. and Harrison, N. (2004). *Organizational Patterns of agile software development*. Prentice-Hall Inc.

[Corbató et al., 1962] Corbató, F. J., Merwin-Daggett, M., and Daley, R. C. (1962). An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference*, AIEE-IRE '62 (Spring), pages 335–344, New York, NY, USA. ACM.

[Corbet and Kroah-Hartman, 2016] Corbet, J. and Kroah-Hartman, G. (2016). Linux kernel development. how fast it is going, who is doing it, what they are doing and who is sponsoring the work.

[Coyne, 1999] Coyne, R. (1999). *Technoromanticism: : Digital Narrative, Holism, and the Romance of the Real*. MIT Press.

[Cramer, 2014] Cramer, F. (2014). What is 'post-digital'? *Post-digital research*, 3(1).

[Cunningham, 2014] Cunningham, W. (2014). Wiki history. Accessed 15.03.2019 `http://wiki.c2.com/?WikiHistory`.

[Daniel et al., 1980] Daniel, S., Ellis, J., and Truscott, T. (1980). Usenet - a general access unix network. Handout.

[Das, 2012] Das, S. (2012). *Your UNIX/Linux: The Ultimate Guide*. McGraw-Hill Education, 3 edition.

[Dasgupta, 2016] Dasgupta, S. (2016). *Computer Science. A very Short Introduction*. Oxford.

[Daylight, 2012] Daylight, E. G. (2012). *The Dawn of Software Engineering. From Turing to Dijstra.* Lonely Scholar Scientific Books.

[de Sousa Santos, 2009] de Sousa Santos, B. (2009). *Una epistemología del sur: la reinvención del conocimiento y la emancipación social*.

[Debian Foundation, 2016] Debian Foundation (2016). Debian constitution. Accessed 17.05.2019 `https://www.debian.org/devel/constitution`.

[DeMarco and Lister, 2013] DeMarco, T. and Lister, T. (2013). *Peopleware. Productive Projects and Teams*.

[Dennis Allison, 1976] Dennis Allison, H. L. . F. (1976). Design notes for tiny basic. *Dr. Dobb's Journal of Computer Calisthenics & Orthodontia*, 1(1):5–8.

[Derrida, 1997] Derrida, J. (1997). *Of Grammatology*. Johns Hopkins University Press.

[Dijkstra, 1972] Dijkstra, E. W. (1972). Notes on structured programming. In Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., editors, *Structured Programming*, chapter Chapter I: Notes on Structured Programming, pages 1–82. Academic Press Ltd., London, UK, UK.

[Dorfman and Thayer, 1999] Dorfman, M. and Thayer, R. H. (1999). *Software Engineering*. Wiley-IEEE Computer Society.

[Eastlake, 1972] Eastlake, D. E. (1972). Its status report. Technical report, Massachusetts Institute of Technology, A.I. Laboratory.

[Eck, 1995] Eck, D. J. (1995). *The most complex machine: A survey of computers and computing.* A K Peters Ltd.

[Edson et al., 1956] Edson, J., Greenstadt, J., Greenwald, I., Jones, F. R., and Wagner, F. V. (1956). *SHARE Reference Manual for the IBM 704*. SHARE user group.

[Edwin T. Layton, 1974] Edwin T. Layton, J. (1974). Technology as knowledge. *Technology and Culture*, 15(1):31–41.

[Ensmenger, 2010] Ensmenger, N. (2010). *The Computer Boys take over.* MIT Press.

[Farías, 2014] Farías, I. (2014). Virtual attractors, actual assemblages: How luhmann's theory of communication complements actor-network theory. *European Journal of Social Theory*, 17(I):24–41.

[Flichy, 2007] Flichy, P. (2007). *The Internet imaginaire.* MIT Press.

[Florin, 1986] Florin, F. (1986). Hackers: Wizards of the electronic age. 26 min. Documentary. Accessed 02.12.2016 `https://www.youtube.com/watch?v=zOP1LNr70aU`.

[Fogel, 2017] Fogel, K. (2017). *Producing Open Source Software. How to Run a Successful Free Software Project.* O'Reilly Media.

[Fores, 1979] Fores, M. (1979). The history of technology: An alternative view. *Technology and Culture*, 20(4):853–860.

[Fowler, 2003] Fowler, M. (2003). Catalog of patterns of enterprise application architecture. Accessed 10.07.2019 `https://martinfowler.com/eaaCatalog/`.

[Frabetti, 2011] Frabetti, F. (2011). Rethinking the digital humanities in the context of originary technicity. *Culture Machine*, 12.

[Frabetti, 2015] Frabetti, F. (2015). *Software Theory. A cultural and philosofical study.* Rowman & littlefield.

[Free Software Foundation, 1989] Free Software Foundation (1989). Gnu general public license, version 1. Accessed 29.03.2017 `https://www.gnu.org/licenses/old-licenses/gpl-1.0.en.html`.

[Free Software Foundation, 2007] Free Software Foundation (2007). Gnu general public license. Accessed 10.07.2019 `https://www.gnu.org/licenses/gpl-3.0.en.html`.

[Fressoli et al., 2014] Fressoli, M., DIas, R., and Thomas, H. (2014). *Innovation and Inclusive Development in the South: A Critical Perspective*, pages 47–65. MIT Press.

[Fuller, 2008] Fuller, M. (2008). Introduction. In Fuller, M., editor, *Software Studies. A lexicon*. The MIT Press.

[Fuller et al., 2017] Fuller, M., Goffey, A., Mackenzie, A., Mills, R., and Sharples, S. (2017). Big diff, granularity, incoherence, and production in the github software repository. In Ina Blom, Trond Lundemo, E. R., editor, *Memory in Motion. Archives, Technology and the Social.* Amsterdam University Press.

[Galloway, 2004] Galloway, A. R. (2004). *Protocol. How controls exists after decentralization*. MIT Press.

[Gamma et al., 1994] Gamma, E., Vlissides, J., Johnson, R., and Helm, R. (1994). *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[Gates, 1976] Gates, B. (1976). An open letter to hobbyists. *Homebrew Computer Club Newsletter*, 2(1).

[Giaglis and Spinellis, 2012] Giaglis, G. M. and Spinellis, D. (2012). Division of effort, productivity, quality, and relationships in floss virtual teams: Evidence from the freebsd project. *Journal of Universal Computer Science*, 18(19).

[Glass, 2002] Glass, R. L. (2002). *Facts and Fallacies of Software Engineering*. Addison Wesley.

[Godfrey and Hendry, 1993] Godfrey, M. and Hendry, D. (1993). The computer as von neumann planned it. *IEEE Annals of the History of Computing*, 15(1).

[Goffey, 2008] Goffey, A. (2008). Algorithm. In Fuller, M., editor, *Software Studies. A Lexicon*. MIT Press.

[Grad, 2002] Grad, B. (2002). A personal recollection: Ibm's unbundling of software and services. *IEEE Annals of the History of Computing*, 24(1):64–71.

[Graham, 2010] Graham, P. (2010). *Hackers & Painters. Big Ideas from the Computer Age*. O'Reilly Media.

[Grassmuck, 2004] Grassmuck, V. (2004). *Freie Software. Zwischen Privat - und Gemeineigentum*. Bundeszentrale für politische Bildung.

[Greenfield et al., 2004] Greenfield, J., Short, K., Cook, S., Kent, S., and Crupi, J. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons.

[Greenwald and Martin, 1956] Greenwald, I. D. and Martin, H. G. (1956). Conclusions after using the pact i advanced coding technique. *J. ACM*, 3(4):309–313.

[Guendert, 2011] Guendert, S. (2011). Mainframe history and the first users' groups (share). Accessed 15.10.2016 `http://enterprisesystemsmedia.com/article/mainframe-history-and-the-first-users-groups-share#&ts=undefined`.

[Gutowski, 2014] Gutowski, H. L. E. (2014). *Geschenke im Netz? Gift Economy und Open-Source*. PhD thesis.

[Hadar et al., 2016] Hadar, I., Levy, M., Ben-Chaim, Y., and Farchi, E. (2016). Using wiki as a collaboration platform for software requirements and design. In Kunifuji, S., Papadopoulos, G. A., Skulimowski, A. M., and Kacprzyk , J., editors, *Knowledge, Information and Creativity Support Systems*, pages 529–536, Cham. Springer International Publishing.

[Hafner and Lyon, 1994] Hafner, K. and Lyon, M. (1994). *Where Wizards Stay up late. The Origins of the Internet.* Touchstone.

[Hall, 2008] Hall, G. (2008). *Digitize this book! the Politics of New Media, or Why We Need Open Access Now*. University of Minessota Press.

[Hannebauer, 2016] Hannebauer, C. (2016). *Contribution Barriers to Open Source Projects*. PhD thesis.

[Hannebauer et al., 2010] Hannebauer, C., Wolff-Marting, V., and Gruhn, V. (2010). Towards a pattern language for floss development. In *Conference on Pattern Languages of Programs*.

[Hansen, 2004] Hansen, M. (2004). Realtime synthesis and the différance of the body: Technocultural studies in the wake of deconstruction. *Culture Machine*, 6(0).

[Hauben and Hauben, 1998] Hauben, M. and Hauben, R. (1998). On the early days of usenet: The roots of the cooperative online culture. *First Monday*, 3(8).

[Hauben et al., 1997] Hauben, M., Hauben, R., and Truscott, T. (1997). *Netizens: On the History and Impact of Usenet and the Internet*. Wiley-IEEE Computer Society.

[Hauben, 1998] Hauben, R. (1998). From the arpanet to the internet. a study of the arpanet tcp/ip digest and of the role of online communication in the transition from the arpanet to the internet. Accessed 29.07.2017 `http://www.columbia.edu/~rh120/other/tcpdigest_paper.txt`.

[Hayles, 1999] Hayles, N. K. (1999). *How We Became Posthuman: Virtual Bodies in Cybernetics, Literature, and Informatics*. University Of Chicago Press.

[Hayles, 2004] Hayles, N. K. (2004). Print is flat, code is deep: The importance of media-specific analysis. *Poetics Today*, 25(1):67–90.

[Hayles, 2005] Hayles, N. K. (2005). *My Mother was a Computer. Digital subjects and literary texts.* The University of Chicago Press.

[Healy and Schussman, 2003] Healy, K. and Schussman, A. (2003). The ecology of open source software development. Technical report, University of Arizona.

[Heart et al., 1970] Heart, F., Kahn, R. E., Ornstein, S., Crowther, W., and Walden, D. (1970). The interface message processor for the arpa computer network. In *Spring Joint Computer Conference 1970*.

[Heidegger, 1977] Heidegger, M. (1977). *The Question Concerning Technology and other Essays.* Garland Publishing Inc.

[Heim, 1999] Heim, M. (1999). *Electric Language: A Philosophical Study of Word Processing*. Yale University Press.

[Hernández et al., 2008] Hernández, J. M., Jiménez, D. M., Barahona, J. M. G., Pascual, J. S., and Robles., G. (2008). *Introduction to free software.* Universitat Oberta de Catalunya.

[Himanen, 1999] Himanen, P. (1999). *The Hacker Ethic and the Spirit of the Information Age*. Random House Trade Paperbacks.

[Homan and Swindle, 1959] Homan, C. and Swindle, J. (1959). *Programmers Manual for the SHARE Operating System*. International Business Machines Corporation, preliminary version edition.

[IBM, 1969] IBM (1969). Ibm archives 1960s. Accessed 12.93.2017 `https://www-03.ibm.com/ibm/history/history/year_1969.html`.

[International Business Machines Corporation, 1960] International Business Machines Corporation (1960). *SOS Reference Manual. SHARE System for the 709*. International Business Machines Corporation.

[Isaacson, 2014] Isaacson, W. (2014). *The Innovators. How a Group of Hackers, Geniuses and Geeks Created the Digital Revolution*. Simon & Schuster.

[Izquierdo-Cortazar et al., 2009] Izquierdo-Cortazar, D., Robles, G., Ortega, F., and Gonzalez-Barahona, J. M. (2009). Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10.

[Johnson, 1998] Johnson, L. (1998). A view from the 1960s: how the software industry began. *IEEE Annals of the History of Computing*, 20(1):36–42.

[Kehoe, 1993] Kehoe, B. P. (1993). *Zen and the Art of the Internet: A Beginner's Guide*. Prentice Hall.

[Kelty, 2008] Kelty, C. M. (2008). *Two Bits: The Cultural Significance of Free Software*. Duke University Press Books.

[Kitchin and Dodge, 2011] Kitchin, R. and Dodge, M. (2011). *Code/Space. Software in everyday life.* MIT Press.

[Kittler, 1997] Kittler, F. A. (1997). There is no software. In Johnston, J., editor, *Literature Media Information Systems: Essays*, pages 147–155. Routledge.

[Kluitenberg, 2011] Kluitenberg, E. (2011). Legacies of tactical media. the tactics of occupation: From tompkins square to tahir.

[Krieger and Belliger, 2014] Krieger, D. J. and Belliger, A. (2014). *Interpreting Networks. Hermeneutics, Actor-Network Theory & New Media.* Transcript.

[Krämer, 1988] Krämer, S. (1988). *Symbolische Maschinen. Die Idee der Formalisierung in geschichtlichem Abriß.*

[Lakhani and Wolf, 2005] Lakhani, K. R. and Wolf, R. G. (2005). Why hackers do what they do: Understanding motivation and effort in free/open source software projects. In *Perspectives on Free and Open Source Software.*

[Latour, 1993] Latour, B. (1993). *We havenerver been modern.* Harvard University Press.

[Latour, 1996] Latour, B. (1996). *Aramis or the love of technology.* Hanvard University Press.

[Lee and LiPuma, 2002] Lee, B. and LiPuma, E. (2002). Cultures of circulation: The imaginations of modernity. *Public Culture*, 14(1).

[Lee, 1992] Lee, J. A. N. (1992). Claims to the term 'time-sharing'. *IEEE Annals of the History of Computing*, 14(1):16–54.

[Lee et al., 1992] Lee, J. A. N., McCarthy, J., and Licklider, J. C. R. (1992). The beginnings at mit. *IEEE Annals of the History of Computing*, 14(1):18–54.

[Leiner et al., 2009] Leiner, B. M., Cerf, V. G., Clark, D. D., Kahn, R. E., Kleinrock, L., Lynch, D. C., Postel, J., Roberts, L. G., and Wolff, S. (2009). A brief history of the internet. *ACM SIGCOMM Computer Communication Review*, 39(5):22–31.

[Levy, 1994] Levy, S. (1994). *Hackers: Heroes of the Computer Revolution.* O'Reilly Media.

[Licklider and Taylor, 1968] Licklider, J. and Taylor, R. W. (1990 (1968)). The computer as a communication device. In *In Memoriam: J. C. R. Licklider 1915-1990.* Digital Equipment Corporation.

[Licklider, 1960] Licklider, J. C. R. (1960). Man-computer symbiosis. *IRE Transactions on Human Factors in Electronics*, HFE-1:4–11.

[Licklider, 1963] Licklider, J. C. R. (1963). Memorandum for: Members and affiliates of the intergalactic computer network.

[Lovink and Scholz, 2007] Lovink, G. and Scholz, T. (2007). *Collaboration on the Fence*, pages 9–25. Autonomedia.

[Lowood, 2001] Lowood, H. (2001). The hard work of software history. *RBM: A Journal of Rare Books, Manuscripts, and Cultural Heritage*, 2(2):141–160.

[Lunenfeld, 2011] Lunenfeld, P. (2011). *The secret war between downloading and uploading. Tales of the computer as a cultural machine.* MIT Press.

[Mackenzie, 2003] Mackenzie, A. (2003). The problem of computer code: Leviathan or common power? Accessed 01.08.2015 `http://www.lancs.ac.uk/staff/mackenza/papers/code-leviathan.pdf`.

[Mackenzie, 2005] Mackenzie, A. (2005). The performativity of code. software and cultures of circulation. *Theory, Culture & Society*, 22(1).

[Mackenzie, 2006] Mackenzie, A. (2006). *Cutting Code. Software and sociality.* Peter Lang Publishing Inc.

[Mackie, 2018] Mackie, E. (2018). Technological disobedience: Ernesto oroza. Accessed 15.12.2019 `https://assemblepapers.com.au/2017/04/28/technological-disobedience-ernesto-oroza/`.

[Mahoney, 2004] Mahoney, M. S. (2004). Finding a history for software engineering. *Annals of the History of Computing, IEEE*, 26(1):8–19.

[Manovich, 2002] Manovich, L. (2002). *The Language of New Media.* The MIT Press.

[Manovich, 2013] Manovich, L. (2013). *Software takes command: extending the language of new media.* Bloomsburry.

[Marill and Roberts, 1966] Marill, T. and Roberts, L. G. (1966). Toward a cooperative network of time-shared computers. In *Proceedings of the November 7-10, 1966, Fall Joint Computer Conference*, AFIPS '66 (Fall), pages 425–431, New York, NY, USA. ACM.

[Marino, 2020] Marino, M. C. (2020). *Critical Code Studies.* MIT Press.

[Marthaller, 2017] Marthaller, J. (2017). Beta phase communities: Open source software as gift economy. *Political Theology*, 18(1).

[McCarthy, 1959] McCarthy, J. (1959). Memorandum to p.m. morse proposing time sharing. Accessed 03.01.2017 `http://www-formal.stanford.edu/jmc/history/timesharing-memo/timesharing-memo.html`.

[McCarthy, 1992] McCarthy, J. (1992). Reminiscences on the history of time-sharing. *IEEE Annals of the History of Computing*, 14(1):19–24.

[McKusick, 1999] McKusick, M. K. (1999). Twenty years of berkeley unix: From at&t-owned to freely redistributable. In *Open Sources. Voices from the Open Source Revolution.*, pages 21–27. O'Reilly Media.

[McVoy, 1993] McVoy, L. (1993). The sourceware operating system proposal. Technical report, Sun Microsystems.

[Medina Cardona, 2014] Medina Cardona, L. F. (2014). En código abierto. *Arcadia .*, (107):43.

[Medina Cardona, 2016] Medina Cardona, L. F. (2016). The free/open source software as hybrid concept: Considerations from a colombian activist perspective. *Media N. Jpurnal of the New Media Caucus*, 12(1):76–81.

[Melahn, 1956] Melahn, W. S. (1956). A description of a cooperative venture in the production of an automatic coding system. *J. ACM*, 3(4):266–271.

[Mestres and Hinojos, 2017] Mestres, A. and Hinojos, M. (2017). *Buscando un corazón al hombre-lata*, pages 9–16. Transit Projectes.

[Metahaven, 2015] Metahaven (2015). *Black Transparency. The Right to Know in the Age of Mass Surveillance.* Sternberg Press.

[Miller and Oldfield, 1956] Miller, Jr., R. C. and Oldfield, B. G. (1956). Producing computer instructions for the pact i compiler. *J. ACM*, 3(4):288–291.

[Moody, 2002] Moody, G. (2002). *Rebel code. Inside linux and the open source revolution.* Basic Books.

[Mozilla Foundation, nd] Mozilla Foundation (n.d.). Mozilla public license version 2.0. Accessed 10.07.2019 `https://www.mozilla.org/en-US/MPL/2.0/`.

[Mulder, 2004] Mulder, A. (2004). *Understanding Media Theory.* V2/Nai Publishers.

[Multicians, 2015] Multicians (2015). Multics - features. Accesed 10.01.2017 `http://www.multicians.org/features.html`.

[Möller, 2006] Möller, E. (2006). *Die heimliche Medienrevolution. Wie Weblogs, Wikis und freie Software die Welt verändern*. Heise.

[Naur and (Eds.), 1969] Naur, P. and (Eds.), B. R. (1969). Sofware engineering. report on a conference sponsored by the nato science committee garmisch, germany, 7th to 11th october 1968. Technical report, NATO.

[Nelson, 1974] Nelson, T. (1974). Computer lib: You can and must understand computers now; dream machines: New freedoms through computer screens— a minority report. Selfpublished.

[Netscape Communications Corporation, 1998] Netscape Communications Corporation (1998). Netscape announces plans to make next-generation communicator source code available free on the net. Accessed 15.04.2018 `https://www.linux-mips.org/archives/linux-mips/1998-01/msg00163.html`.

[Network Working Group, 1999] Network Working Group (1999). Request for comments: 2555: 30 years of rfcs. Accessed 29.07.2017 `https://tools.ietf.org/html/rfc2555`.

[Niquette, 2006] Niquette, P. (2006). Softword: Provenance for the word 'software'. Accessed 20.10.2016 `http://niquette.com/books/softword/tocsoft.html`.

[Norbert Wiener, 1993] Norbert Wiener, W. a. i. b. S. J. H. (1993). *Invention: The care and feeding of ideas*. MIT Press.

[Open Source Initiative, 2007] Open Source Initiative (2007). The open source definition. Accessed 13.04.2018 `https://opensource.org/osd`.

[Osio, 2016] Osio, U. R. (2016). *Filosofía y software: La cultura digital detrás de la pantalla*. Universidad de Lima.

[Pabón et al., 2016] Pabón, O. H. P., González, F. A., Aponte, J., Camargo, J. E., and Restrepo-Calle, F. (2016). Finding relationships between socio-technical aspects and personality traits by mining developer e-mails. In *2016 IEEE/ACM Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 8–14.

[Pfaffenberger, 2003] Pfaffenberger, B. (2003). A standing wave in the web of our communications": Usenet and the socio-technical construction of cyberspace values. In Lueg, C. and Fisher, D., editors, *From Usenet to CoWebs: Interacting with Social Information Spaces*, Computer Supported Cooperative Work, pages 20–41. Springer-Verlag London, 1 edition.

[Poonam and Yasser, 2018] Poonam, R. and Yasser, C. M. (2018). An experimental study to investigate personality traits on pair programming efficiency in extreme programming. In *5th International Conference on Industrial Engineering and Applications*, pages 95–99.

[Raymond, 1999] Raymond, E. S. (1999). The revenge of the hackers. In *Open Sources. Voices from the Open Source Revolution*. O'Reilly Media.

[Raymond, 2001] Raymond, E. S. (2001). *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, 1 edition.

[Raymond, 2003] Raymond, E. S. (2003). *The Art of UNIX Programming*. Addison-Wesley Professional, 1 edition.

[Rheingold, 2007] Rheingold, H. (2007). *Technologies of Cooperation*, pages 29–63.

[Ritchie, 1984] Ritchie, D. M. (1984). The unix system: The evolution of the unix time-sharing system. *AT T Bell Laboratories Technical Journal*, 63(8):1577–1593.

[Ritchie, 1993] Ritchie, D. M. (1993). The development of the c language. *SIGPLAN Not.*, 28(3):201–208.

[Ritchie and Thompson, 1974] Ritchie, D. M. and Thompson, K. (1974). The unix time-sharing system. *Commun. ACM*, 17(7):365–375.

[Roberts, 1988] Roberts, L. (1988). The arpanet and computer networks. In Goldberg, A., editor, *A History of Personal Workstations*, pages 141–172. ACM, New York, NY, USA.

[Roberts, 1967] Roberts, L. G. (1967). Multiple computer networks and intercomputer communication. In *Proceedings of the First ACM Symposium on Operating System Principles*, SOSP '67, pages 3.1–3.6, New York, NY, USA. ACM.

[Rodríguez-Bustos and Aponte, 2012] Rodríguez-Bustos, C. and Aponte, J. (2012). How distributed version control systems impact open source software projects. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 36–39.

[Russell, 2013] Russell, A. L. (2013). Osi: The internet that wasn't. Accessed 15.06.2018 `http://spectrum.ieee.org/tech-history/cyberspace/osi-the-internet-that-wasnt`.

[Salus, 1994] Salus, P. H. (1994). *A Quarter Century of UNIX.* Addison-Wesley Publishing Company.

[Salus, 2008] Salus, P. H. (2008). *The Daemon, the Gnu, and the penguin. How free and open software is changing the world.* Reed Media Services.

[Sen et al., 2008] Sen, R., Subramaniam, C., and Nelson, M. L. (2008). Determinants of the choice of open source software license. *Journal of Management Information Systems*, 25(3):207–239.

[Sennett, 2008] Sennett, R. (2008). *The Craftsman.* Penguin Books.

[Share Inc., 1977] Share Inc. (1977). *SHARE PROGRAM LIBRARY AGENCY. User's Guide and Catalog of Programs.* SHARE.

[Shaw, 2008] Shaw, D. B. (2008). *Technoculture.* Berg.

[Shell, 1959] Shell, D. L. (1959). The share 709 system: A cooperative effort. *J. ACM*, 6(2):123–127.

[Shihab et al., 2010] Shihab, E., Bettenburg, N., Adams, B., and Hassan, A. E. (2010). On the central role of mailing lists in open source projects: An exploratory study. In Nakakoji, K., Murakami, Y., and McCready, E., editors, *New Frontiers in Artificial Intelligence*, pages 91–103, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Sinha and Rajasingh, 2014] Sinha, T. and Rajasingh, I. (2014). Investigating substructures in goal oriented online communities: Case study of ubuntu irc. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 916–922.

[Sommerville, 2011] Sommerville, I. (2011). *Software Engineering (9Th Edition).* Pearson, 9th edition.

[Sowe and Zettsu, 2013] Sowe, S. K. and Zettsu, K. (2013). Collaborative development of data curation profiles on a wiki platform: Experience from free and open source software projects and communities. In *Proceedings of the 9th International Symposium on Open Collaboration*, WikiSym '13, pages 16:1–16:8, New York, NY, USA. ACM.

[Spehr, 2003] Spehr, C. (2003). *Gleicher als andere. Eine Grundlegung der freien Kooperation.*

[Spinellis et al., 2016] Spinellis, D., Louridas, P., and Kechagia, M. (2016). The evolution of c programming practices: A study of the unix operating system 1973–2015. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 748–759, New York, NY, USA. ACM.

[Squire and Smith, 2015] Squire, M. and Smith, A. (2015). *The diffusion of pastebin tools to enhance communication in FLOSS mailing lists.*, volume 451 of *IFIP Advances in Information and Communication Technology*. Springer New York LLC, Elon University.

[Stach, 2001] Stach, H. (2001). *Zwischen Organismus und Notation. Zur kulturellen Konstruktion des Computer-Programms.* Deutscher Universitäts·Verlag.

[Stallman, 1983] Stallman, R. (1983). Initial announcement (gnu). Accessed 29.03.2017 `https://www.gnu.org/gnu/initial-announcement.en.html`.

[Stallman, 1985] Stallman, R. (1985). The gnu manifesto. Accessed 29.03.2017 `https://www.gnu.org/gnu/manifesto.en.html`.

[Stallman, 2002] Stallman, R. M. (2002). *Free Software, Free Society: Selected Essays of Richard M. Stallman.* GNU Press.

[Steel, 1957] Steel, Jr., T. B. (1957). Pact ia. *J. ACM*, 4(1):8–11.

[Stiegler, 1998] Stiegler, B. (1998). *Technics and Time, 1. The Fault of Epimethus.* Stanford University Press.

[Stiegler, 2006] Stiegler, B. (2006). Anamnesis & hypomnesis: The memories of desire. In *Technicity*. Literaria Pragensia.

[Suber, 1988] Suber, P. (1988). What is software? *Journal of Speculative Philosophy*, 2(2):89–119.

[Suber, 2012] Suber, P. (2012). *Open Access*. MIT Press.

[Taubert, 2006] Taubert, N. C. (2006). *Produktive Anarchie? Netzwerke freier Softwareentwicklung.* Transcript-Verlag.

[Terceiro et al., 2012] Terceiro, A., Souza, R., and Chavez, C. (2012). Patterns for engagement in free software projects. In *Proceedings of the IX Latin American Conference on Pattern Languages of Programming*, pages 1–20.

[The Jargon File (version 4.4.7), nd] The Jargon File (version 4.4.7) (n.d.). (hacker entry). Accessed 10.12.2016 `http://www.catb.org/jargon/html/H/hacker.html`.

[The Open Source Initiative, nd] The Open Source Initiative (n.d.). The 3-clause bsd license. Accessed 10.07.2019 `https://opensource.org/licenses/BSD-3-Clause`.

[Toomey, 2011] Toomey, W. (2011). The strange birth and long life of unix. *IEEE Spectrum*, 48(12):34–55.

[Torvalds, 1991] Torvalds, L. (1991). First linux announcement (what would you like to see most in minix?). USENET. Accessed 15.03.2018 `https://groups.google.com/forum/#!topic/comp.os.minix/dlNtH7RRrGA%5B301-325%5D`.

[Trogemann, 2005] Trogemann, G. (2005). Experimentelle und spekulative informatik. In Pias, C., editor, *Zukünfte des Computers*, pages 109–132. Diaphanes.

[Valencia, 2017] Valencia, J. C. (2017). *Hackear el caballo de Troya: la colonialidad del software, el Antropoceno y sus alternativas*, pages 65–81. Editorial Pontificia Universidad Javeriana.

[von Neumann, 1993] von Neumann, J. (1993). First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75.

[Walden and Guys, 2014] Walden, D. and Guys, I. S. (2014). The arpanet imp program: Retrospective and resurrection. *IEEE Annals of the History of Computing*, 36(2):28–39.

[Wang and Perry, 2015] Wang, Z. and Perry, D. E. (2015). Role distribution and transformation in open source software project teams. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 119–126.

[Wardrip-Fruin, 2009] Wardrip-Fruin, N. (2009). *Expressive Processing*. MIT Press, Massachusetts.

[Wardrip-Fruin and Montfort, 2003a] Wardrip-Fruin, N. and Montfort, N. (2003a). Introduction - as we may think. In Wardrip-Fruin, N. and Montfort, N., editors, *The New Media Reader*, pages 35,36. MIT Press.

[Wardrip-Fruin and Montfort, 2003b] Wardrip-Fruin, N. and Montfort, N. (2003b). Introduction - man-computer symbiosis. In Wardrip-Fruin, N. and Montfort, N., editors, *The New Media Reader*, page 73. MIT Press.

[Wardrip-Fruin and Montfort, 2003c] Wardrip-Fruin, N. and Montfort, N. (2003c). Introduction - man, machines and the world about. In Wardrip-Fruin, N. and Montfort, N., editors, *The New Media Reader*, pages 65,66. MIT Press.

[Wark, 2004] Wark, M. (2004). *A hacker manifesto*. Harvard University Press.

[Warren-Jr., 1976] Warren-Jr., J. C. (1976). Correspondence. In *SIGPLAN Notices*, volume 11, page 1. ACM, New York, NY, USA.

[Warsta and Abrahamsson, 2003] Warsta, J. and Abrahamsson, P. (2003). Is open source software development essentially an agile method? In *3rd Workshop on Open Source Software Engineering*.

[Weber, 2004] Weber, S. (2004). *The success of open source*. Harvard University Press.

[Wiener, 2003] Wiener, N. (2003). Men, machines and the world about. In Wardrip-Fruin, N. and Montfort, N., editors, *The New Media Reader*, pages 67–72.

[Wilbur, 1997] Wilbur, S. P. (1997). An archaeology of cyberspaces: Virtual, communuty, identity. In Porter, D., editor, *Internet Cultures*, pages 5–22. Routledge.

[Wilson and Aranda, 2011] Wilson, G. and Aranda, J. (2011). Empirical software engineering: As researchers investigate how software gets made, a new empire for empirical research opens up. *American Scientist*, 99(6):466–473.

[Wirth, 2008] Wirth, N. (2008). A brief history of software engineering. *Annals of the History of Computing, IEEE*, 30(3):32–39.

[Wirth, 2005] Wirth, U. (2005). Die epistemologische rolle von links in wissensprozessen. eine mediengeschichtliche rekonstruktion. In Gendolla, P. and Schäfer, J., editors, *Wissensprozesse in der Netzwerkgesellschaft*, page 43–54. Transcript.

[Xiao et al., 2007] Xiao, W., Chi, C., and Yang, M. (2007). On-line collaborative software development via wiki. In *Proceedings of the 2007 International Symposium on Wikis*, WikiSym '07, pages 177–183, New York, NY, USA. ACM.

[Yu et al., 2011] Yu, L., Ramaswamy, S., Mishra, A., and Mishra, D. (2011). Communications in global software development: An empirical study using gtk+ oss repository. In Meersman, R., Dillon, T., and Herrero, P., editors, *On the Move to Meaningful Internet Systems: OTM 2011 Workshops*, pages 218–227, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Ziegenbalg et al., 2016] Ziegenbalg, J., Ziegenbalg, O., and Ziegenbalg, B. (2016). *Algorithmen von Hammurapi bis Gödel: Mit Beispielen aus den Computeralgebrasystemen Mathematica und Maxima*. Springer Spektrum, 4 edition.

[Zuboff, 2015] Zuboff, S. (2015). Big other: Surveillance capitalism and the prospects of an information civilization. *Journal of Information Technology*, 30(1):75–89.

Luis Fernando Medina Cardona has been, since 2010, an associate professor at the school of cinema and television at the Universidad Nacional de Colombia (Bogotá, Colombia) where he teaches art and new technologies and research methods. He has been a fellow at the Kunsthochschule für Medien Köln (Cologne, Germany, 2012), and guest professor at the design and creation master's program at the Universidad de Caldas (Manizales, Colombia, 2018).

His work alternates between academic research and cultural practices, involving free software as a tool and a culture, collaborative methodologies, alternative media, digital humanities, and open science, all under the umbrella of a so-called open collaboration metaphor. He has participated in various culture projects concerning these topics and has worked as advisor to the city of Bogotá in art and free technologies issues; particularly, in the open radio broadcasting project "CKWEB".

In the Universidad Nacional de Colombia, he was responsible for the first course on arts of free technologies and advocating open access and open science practices in the school of arts. For the previous, he received the teaching excellence award (2013). He was also granted a doctoral scholarship by the German Academic Exchange Service DAAD (2014). He is a member of the Colombian network for digital humanities and one of founders of the open science working group in Colombia. Currently, he is the founder and leader of the research group "espacio de producción abierta de medios <\espam >" (space for open media production) at the Universidad Nacional de Colombia.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Dissertation selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe.

Bogotá, 11. Februar 2021.

Luis Fernando Medina Cardona