ABSTRACT

KUDENOV, PETER ALEXANDER. Software Engineering as a Problem-Oriented Sociotechnical Regime. (Under the direction of Drs. David Rieder and Helen Burgess).

This project describes software engineering as a problem-oriented sociotechnical regime, developing a Deleuze and Guattarian theoretical framework to explain how software is the culmination of a progressive series of problematic encounters between axiomatics—which are the formalized practices which design, implement, test, and maintain software—and the states of affairs for which, and in which, the software is produced. The purpose of this project, and its use of Deleuze and Guattarian theory, is to connect software as a type of media to the practices which produce it, thereby concretizing the reasoning behind its concepts and implementation in the world at large, essentially the transition from the virtual to the actual. This was done in response to media studies' scholarship's lack of engagement with the patterns and practices of software's implementations—the work and the ways software engineers solve problems—to connect a crucially important media technology to the practices and perspectives that design and implement it. This dissertation has three major parts. The first part defines the problem-oriented sociotechnical regime, laying the theoretical framework for the rest of the dissertation. It describes how social and technical factors and resources relate to an orienting problematic, which is the combination of a problem and its conditions in a Deleuzian sense. Regimes represents a type of stable identity that has cohered around a problematic and the axiomatics designed to implement solutions to the problem related to its conditions. The second part of this dissertation examines the impetus for 'software engineering' to exist, namely as a response to the 'software crisis' of the 1950s and 60s, by looking at the history of early software development, the inception of Computer Science, managerial practices, and the convening of the 1968 NATO Conference on Software Engineering. Software engineering—still a polemical combination of

المنارة للاستشارات
www.manaraa.com

terms—was intended to standardize the implementation of software using engineering concepts and practices, thereby making it more predictable and reliable. It is related to but distinct from Computer Science and managerial practices. The third part of this dissertation examines the consequences of communicating with a sociotechnical regime when such communication crosses signifying and asignifying boundaries, a process referred to here as transduction. Software is a diagrammatic, asignifying computational product that must interact with a signifying, human world. Its design requires that signifying values are translated into asignifying diagrammatical processes. Using Guattarian mixed semiotics, transduction describes part of the issue of making reliable software—in essence, the problem of communicating problems—when those understandings shed meanings as they are transduced from signifying domains into asignifying domains, and vice versa. Taken together, this project describes software engineering as a sociotechnical regime, which has acquired its own identity and axiomatics, while providing a basis for interpreting 'software' and the practices that instantiate it for future media studies' scholarship.

المنارة للاستشارات
www.manaraa.com

© Copyright 2019 by Peter A. Kudenov

All Rights Reserved

www.manaraa.com

المنارة للاستشارات
www.manaraa.com

Software Engineering as a Problem-Oriented Sociotechnical Regime

by
Peter Alexander Kudenov

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Communications, Rhetoric and Digital Media

Raleigh, North Carolina
2019

APPROVED BY:

Dr. David Rieder Committee Co-Chair	Dr. Helen Burgess Committee Co-Chair
Dr. Stephen Wiley	Dr. Timothy Menzies

المنارة للاستشارات

www.manaraa.com

ProQuest Number: 27529103

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 27529103

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

للاستشارات المنارة

www.manaraa.com

DEDICATION

This project is dedicated to Mary, my wife and partner, whose mettle, understanding and patience made it possible to pursue—and persist with—this work. Further, my oldest son, David, and newest, Carter, gave me the impetus even on difficult days to keep at it. And finally, to my parents, whose support and enthusiasm over the years has been priceless and treasured.

المنارة للاستشارات
www.manaraa.com

BIOGRAPHY

Peter Kudenov holds a Bachelor's (2009) and a Master's degree (2013) in English. Professionally, he has been designing and implementing a variety of technology solutions for clients as a software developer and engineer since 2001. His longstanding interests in technology, digital media, and rhetorical theory and practices lead him to choose NC State's Communications, Rhetoric, and Digital Media program as a natural evolution to further his interests in media studies and the historical practices associated with the development of technology.

www.manaraa.com

ACKNOWLEDGMENTS

This project would not have been possible without the support of David Rieder, whose patience and enthusiasm for my topic provided the encouragement needed to move forward with the work. Stephen Wiley provided detailed feedback and always challenged me to think through broader issues ("whose problems?"), providing a basis for moving forward with my work into the world. Helen Burgess helped me move the project over the finish line, and Timothy Menzies offered a counterpoint and reality to the abstractions that I always found invaluable. Thank you.

المنارة للاستشارات
www.manaraa.com

TABLE OF CONTENTS

LIST OF TABLES vi
LIST OF FIGURES vii

Chapter 1: Software Engineering as a Problem-Oriented Sociotechnical Regime........ 1
- There is No Software? No: There Are No Typical Software Projects 3
- On Sociotechnical Regimes 8
- On Software and Software Engineering, Programming, and Source Code 12
- The Visibility of Software Requires Software Engineering 30

Chapter 2: Problem-Oriented Sociotechnical Regimes........ 33
- Problematics........ 38
- Problem-Orientation 46
- Sociotechnical Regimes 53
- What is a Sociotechnical Regime?........ 54
- How does a Sociotechnical Regime Work?........ 63
  - Events........ 64
  - Plateaus and Rhizomes. 67
  - Smooth and Striated Spaces........ 70
- Describing Software Engineering as a Sociotechnical Regime. 73

Chapter 3: Tracing the Emergence of Software Engineering as a Sociotechnical Regime 78
- Software Engineering........ 82
- Early Software 83
- 1968, Formalization of Computer Science 89
- Software in Crisis: Intersections of Computation and Management Practices........ 104
- Emergence of Software Engineering 117
- Software Engineering as a Plateau........ 134

Chapter 4: Transduction and Mixed Semiotics of Software Engineering........ 137
- Asignification........ 140
- Asignification Defined........ 141
- Collection Assemblage of Enunciation........ 144
- Diagrammatism in McLuhan and Kittler: Consistency, Automation, and the Digital .. 149
- Diagrammatic Processes, Digital Technology........ 154
- Transduction 160
  - Necessary limitations. 167
  - Potentials from limits. 170
- Problematics and Transduction........ 184

Chapter 5: On Sociotechnical Regimes, Software Engineering, and Transducive Practices........ 188
- Contributions........ 192
- Summaries........ 195
- Applications 198
- Future Directions 200

References........ 202

المنارة للاستشارات
www.manaraa.com

LIST OF TABLES

Table 3.1 Significant Events in the Inception of Computer Science 92

Table 3.2 A slightly modified first C program from Deitel & Deitel’s (1994) textbook, adjusted to include the stdio.h header so that standard input and output subroutines, like “printf,” are available to the program, making it buildable (p. 24). 122

Table 4.1 Typical steps for a design process (reproduced from Day, 2017, p. 43). 183

المنارة للاستشارات

www.manaraa.com

LIST OF FIGURES

Figure 3.1 The Curriculum Map of Math ("M"), Basic ("B"), Intermediate ("I"), and Advanced ("A") courses established by Atchison et al. in 1968. 99

Figure 3.2 The formula for cost analysis forwarded at the 1962 RAND Symposium, "On Programming Languages" (Patrick et al., 1962, p. 25). 113

Figure 4.1 Guattarian Assemblages of Enunciation (reproduced from Guattari, 2011, p. 57). 147

Figure 4.2 Guattarian Fields of Consistency (reproduced from Guattari, 2011, p. 51). 153

Figure 4.3 Three Limiting Fields (reproduced from Guattari, 2011, p. 60) 166

المنارة للاستشارات

www.manaraa.com

Chapter 1: Software Engineering as a Problem-Oriented Sociotechnical Regime

This dissertation project is a direct response to an issue brought forth by Wendy Chun (2011), who argued that media studies' engagements with software made it more ephemeral and therefore less knowable, causing software to become a fetishistic medium. In response, this project seeks to de-fetishize source code, programming, and software by looking broadly at software engineering through a Deleuze and Guattarian framework. De-fetishizing software has significant implications for media theorists but has seemingly not been picked up as an area of study by humanist scholars. Chun argues that attempts to study source code that try "to map and know the workings of [a] machine" as a basis for understanding it and theorizing it as a media technology tend toward a type of reductionism that equates source code to action, i.e., as a "magical entity" and "source of causality" (p. 51). To combat this, Chun asserts that to "make our computers more productively spectral" scholars must embrace "the unexpected possibilities of source code" by treating it as a fetish (p. 20), which "allows one to visualize what is unknown" by "[substituting] images for causes" (pp. 50-51). Chun argues against the belief that source code is what it is, against the position Ellen Ullman took, i.e., that source code's "entire meaning is its function" (Ellen qtd in Chun), because it can be read by many human and nonhuman agents and understood in multiple ways. Consequently, due to its readability and ability to convey multiple types of information, source code *is a medium* in the "full sense of the word," because "it channels the ghost that we imagine runs the machine" (pp. 49-50): "the power" of (or behind) source code, she asserts, "lies elsewhere … in *social and machinic relations*" (emphasis added, p. 51). This is true, in the sense that software does not emerge into existence without some impetus for that existence. Software is designed into existence. Arguments about what software is need to incorporate an understanding of how software is

المنارة للاستشارات
www.manaraa.com

made, and consequently, what software engineering *does*; doing so will de-fetishize software, allowing it to become knowable and seeable in ways media studies has typically not integrated into its theoretical milieus. As evidenced later in this chapter, media theorists studying software have generally *not* moved past the primacy of code and the concept of the programmer, even in automated and machine learning systems, and have not given an account of the ontological and epistemological values that broadly deploy and execute software. My argument is that source code and programming are but *one process* in a *set of processes* that produces software, and that looking past the programmer and source code to the practices of software engineering integrates the world of **social and machinic relations** to which Chun alluded into future definitions of software: programming is but one process of many that brings source code into existence. Software is not a ghostly medium if its exigencies and ontologies of design, management, and funding are incorporated into its definition. So, while Chun (2011) does look at certain types of social and machinic relations in *Programmed Visions*, the reasoning for the processes leading to the 'software' she and others use, like Alexander Galloway (2012) in *Interface Effect*, are largely taken for granted in the larger and ongoing discussion about power and its consequences. Software engineering is the field of praxis that produces most modern software, and understanding it is a way to integrate knowledge that de-emphasizes the ways of knowing and ontological processes of proposal writing, requirements gathering, specification validation, development, testing, and documentation that lead to software and source code as mediating things in the world. Rather than focus on power per se, I examine how and why software is produced in the way it is produced by looking at how the discipline of software solves problems as a way to engage media theorists with an aspect of software media that is underdeveloped.

المنارة للاستشارات
www.manaraa.com

The way to account for the reasoning of software, the 'why' of the 'how' software is made, is to set aside the concept of power and instead examine the *problematics*—which to Deleuze (1994) are the combination of a problem with its conditions—leading to the *production of the disciplines themselves* which are dedicated to solving problems computationally, e.g., those of Computer Science and Software Engineering. What produces the identity of a software engineer, what does it mean to 'engineer' software, and why is the conjunction of 'software' with 'engineer' both polemical and desirable? An analysis of problematics is where Deleuze and Guattari's independent and conjunctive works bloom, with its emphasis on ontologies of becoming and of rhizomatic associations stemming from events, which I mobilize to produce an analytical framework capable of accounting for why software engineers produce software in ways that they do. To this end, this dissertation develops the concept of the problem-oriented sociotechnical regime to frame a historical analysis of the factors leading to the inception and evolution of software engineering, which then examines the difficulties inherent to the mixed semiotic interactions of transducing problems into software.

There Is No Software? No: There Are No Typical Software Projects

Before turning to the concept of the sociotechnical regime, it is necessary to describe how software is generally produced. Considering the processes of its production refutes narratives that software is ephemeral or unknowable. It must be emphasized that there are no 'typical' software development projects, and despite programming and source code being emblematic of software development, they represent, as Brooks (1995) argued, about 1/6th of the work in bringing software development to fruition in the form of a deployable, usable product. Brooks oversaw the development of IBM's OS/360 in the 1960s, which became one of the most

المنارة للاستشارات
www.manaraa.com

expensive software development efforts at the time and would come to realize that software development is confounding in many ways. While one might assume that if a product were running behind, adding more developers would speed up the development process, Brooks found it held the project up in several crucial ways. At the time, more developers meant, blithely, more sick days, more scheduling headaches, less effective communication among the team members, and slower overall progress; counter-intuitively, fewer developers produced software faster. From a 'programming' perspective—the extent to which a developer just writes code in their day-to-day job—Brooks' ratio arguably changes depending on the kind of Software Development Life Cycle (SDLC) their organization adopts, and the level of seniority and experience they have. Junior developers tend to do less planning and design work, while senior developers and architects might regularly meet with stakeholders, write requirements documents, map out designs, and assign tasks and prioritize work for junior team members. Brooks' ratio shows that software is more than its programming and source code and is rather an aggregation of efforts.

Software development is far more than programming and compiling source code cloned from git repositories. While Brooks found that planning, documentation, testing, and maintenance took more time than implementation work, the most valuable insight is the realization that software development is fundamentally communicative across human and technical domains. Planning ideally takes place prior to implementation, and includes tasks like needs assessments, requirements gathering, and design work, which are all processes that are performed to such an extent that the problem someone is having is understood well enough that a tenable solution can be found. During planning, project managers, architects, and senior developers talk to clients and stakeholders, turning the information they gather into documentation like design documents or user stories, so that the solution becomes a kind of

المنارة للاستشارات
www.manaraa.com

anticipated outcome. Those requirements are then divided up and communicated with team members—like artists and programmers—who take them and start turning them into something that performs work. User interfaces are created and connected programmatically to the business logic they are designed to use, and the work is managed over time by senior developers and project managers. Ongoing work is checked against the design documents to ensure conformity, and meetings are held when issues arise that slows the work down to determine if the tasks can be divided and completed by other team members. Testing can mean several things in a software development effort, but generally it is a means to determine if the solution meets the requirements stated at the outset: does the software do what the customer wants it to do? Anytime software fails to meet its requirements, implementations are fixed, possibly leading to new programming work, which then elicits more testing. A major caveat with bug fixing is the additional testing (and sometimes, documentation) it requires: a bug fix might introduce a regression, which occurs when the fix for one issue causes a new issue; a fix can break other parts of an application. Software enters a 'maintenance mode' after it has been delivered to a customer, and many bugs are often fixed after a product has entered the market. Maintenance modes can often last years, if not longer, depending on the needs of the customer, the nature of the software, and the budget allocated for its lifecycle.

While programming is how software is 'written,' software cannot be planned, tested, or maintained without multiple layers of communication: many words are spoken and written to facilitate a software project. As Brooks noted, the implementation phase of an application, while important, is one part of a larger set of processes that are largely driven by people talking to each other. This communicative impetus is part of the reason why software, popularly, is never finished; unlike a corporeal structure like a bicycle frame, which can only be modified so much

المنارة للاستشارات
www.manaraa.com

before it becomes 'not-bicycle frame' (or perhaps, 'was-bicycle frame'), software can be modified and scoured in myriad ways while still maintaining cohesion because improvements can always be integrated and bugs can always be fixed as long as they are brought to the attention of those charged with maintaining it. For example, Microsoft recently increased the number of Fiber Local Storage slots in the Windows 10 Kernel which is notable because the feature lives in the code the kernel uses to manage threading, which allows many processes to run simultaneously on a computer. As the change touches virtually every application Windows runs, it will require long-term testing to verify that it works properly without unduly affecting other parts of the Windows Operating System (OS). Throughout it all, Microsoft has communicated with stakeholders, like its developers, its testers and Windows Insiders' members, and customers potentially affected by (and benefiting from) the changes. It was implemented when a program manager in Microsoft made the case for the change to the Kernel team, attracting enough supporters to the argument for a risky change. He made the case because of the feedback he received from professional audio producers and musicians on a pro-audio forum because the need existed, and it was an issue that would enable Windows to be used far into the future. In a real sense, the communication around software and about it enables it to be produced.

The areas Brooks' identified—planning, implementation, testing, and maintenance—have been codified as SDLCs, which have emerged over the years since IBM's OS/360 debacle to operationally model idealized development efforts. SDLCs are an attempt to reduce human fallibilities by standardizing the ways software is made by addressing changing development environments, technologies, and requirements. The Waterfall SDLC, forwarded by Winston W. Royce in a piece published in 1970, is commonly considered to be the first models to become

المنارة للاستشارات
www.manaraa.com

widely adopted by industry after it was standardized by the Department of Defense in 1985 (DOD, "Defense Systems Software Development," DOD-STD-2167A). Waterfall development efforts flow from top-to-bottom (like a waterfall), such that a development effort begins with a 'requirements' phase and proceeds, in turn through design, implementation, verification, and maintenance phases. Waterfall works well for smaller projects but leads to problems when changes to specifications are introduced mid-course, because the model assumes that all the requirements have been gathered and accounted during a project's inception. Brooks' experience shows—and as Chapter 3 will detail—that large projects have many unforeseen requirements. Contemporary SDLCs still include Brooks' phases but change the way they operate. Agile or Incremental SDLCs tend to integrate documentation and testing into their implementation phase with ideas like self-documenting code and iterative development to, ideally, make them highly responsive to requirements changes. In Agile, requirements are revisited repeatedly. Scott W. Ambler (2018) explains that "Agilists [sic] take an evolutionary, iterative and incremental, approach to development," because "requirements are identified throughout most of [a] project." The work to implement software relies on concurrent or successive efforts to plan, test, and maintain the product.

Every software development effort is atypical in its own way because software is far more than its programming and source code. The source code one reads only offers a small glimpse of how the software emerges, ontologically, into becoming. A program represents an aggregation of efforts in human and technical domains. The fascinating aspect of software is that it is the crux of a natural rhizome, in a Deleuze and Guattarian sense, because each effort is atypical, flowing from requirements and problems that are often incompletely grasped. Where SDLCs like Waterfall or Agile act as models for development efforts to follow in the hopes of

المنارة للاستشارات
www.manaraa.com

increasing the predictability and reliability of the results (topics discussed in chapters 3 and 4), the reality is that models are abstractions and the reality of the day-to-day effort to engineer software is full of sick days, hurt feelings, broken tools, bad routers, power struggles, and miscommunications. Software is a representation of a set of processes which, through aggregated efforts, culminate in a process that works on a problem computationally, and each 'becoming' is a unique negotiation of human and technical communication.

On Sociotechnical Regimes

To define software engineering as a problem-oriented sociotechnical regime and in turn expand the scholarly focus from source code or software toward the social and machinic relations in the processes used to it, this dissertation makes three supporting arguments. In Chapter 2, "Problem-Oriented Sociotechnical Regimes," I develop the idea of a sociotechnical regime to describe how problems organize and coalesce human and non-human agents into sets of technical and social relations and practices that form relatively stable recursive values, identities, and products. From a Deleuze and Guattarian perspective, the concept of the sociotechnical regime addresses the aspect of Chun's (2011) argument about 'sourcery' and its focus on 'source code' by allowing scholarly focus to pivot toward the broad collection of social and machinic processes leading to the production of software. For their recognition of the role problems play in materialist 'becomings' as open-ended processes, I elaborate on concepts from Deleuze and Guattari's individual and collaborative work to create the framework of a sociotechnical regime, which allows me to describe 'software engineering' as a distinct, but co-dependent praxis of Computer Science and managerial considerations. A problem-oriented sociotechnical regime has a peculiar orientation to the world, and the idea is important because it

المنارة للاستشارات
www.manaraa.com

allows one to build context around and demystify the solutions they produce, the practices and the products of those practices derived from estimable and reproduceable axiomatics by a regime. Sociotechnical regimes are part of my response to Chun's concern about the fetishization of code by placing 'source code' in context with the practices and actors involved in its creation. Software is less about its ghostly visage and more about the processes that created it. Technological development is iterative and additive in nature, and sociotechnical regimes do not spontaneously emerge into the world. Problems become visible as soon as the right techniques and processes exist to expose them within their conditions. Therefore, sociotechnical regimes are organized material responses to a problematic (a problem and its conditions, according to Deleuze) that seek to solve that problem while negotiating on-going changes to its conditions. Sociotechnical regimes, like Computer Science or managerial praxis generally, persist through time as they iterate over their solutions to the problems embedded in their relative conditions. The concept allows a definition of software engineering to have its own identity as a discipline while also encapsulating practices and techniques evident in other regimes.

After the concept of the sociotechnical regime is adequately described in Chapter 2, it becomes possible in Chapter 3, 'Software Engineering as Sociotechnical Regime,' to look at the historical factors and problematics leading to the development of software engineering as a distinct regime from the neighboring communities of practice of Computer Science and managerial praxis. The historical factors point to how software engineering became a sociotechnical regime comprising a distinct expression of a social and machinic epistemology that is principally unstable in that it resides in a constant state of tension between Computer Science and corporatized managerial praxis. Mapping 'software engineering' as a sociotechnical regime incorporates many patterns and practices that allow source code to be demystified and

للاستشارات المنارة
www.manaraa.com

revalued. The historical evidence in the organization of software engineering will show that code and coding are one of many practices and products in the work *surrounding* computation, and while important, programming and source code are always an expression of a problematic, which is a problem and its conditions. The history of software engineering reveals more about the nature of software when the practices used to create and deploy it are shown to connect to practices inherent in Computer Science and managerial praxis. Software is theory and practice and is an immanent material expression of social and machinic relations; it maintains its 'ghostly' qualities only when the broad array of social and machinic relations leading to its creation are uninterrogated.

Chapter 4, 'Unstable Intersections,' completes the circuit of the problem-oriented sociotechnical framework by exploring and defining transduction, which I define as the interfacing operation a sociotechnical regime uses to relate to the world and itself. The chapter argues for an interpretation of Guattari's concept of mixed semiotics to show how software engineering, a sociotechnical regime premised on a principally asignifying plane of reference comprised of functions and propositions, relates to the signifying content of human expressions. The chapter performs three tasks: it defines Guattari's concept of asignification by locating its role in enunciative acts, e.g., the mixed semiotics of assemblages of enunciation, and explores its relationships to the works of McLuhan and Kittler to show how his orientation toward diagrammatism positions him as an ideal basis for posthumanist inquiry; it defines the concept of transduction as it regards the actual movement of values and meaning into and out of asignifying and signifying processes and therefore into and out of sociotechnical regimes; and it clarifies the role of signification in a sociotechnical regime by elucidating the relationship a signifier has with the boundaries between diagrammatic and signifying processes. From these moves, I will explain

المنارة للاستشارات
www.manaraa.com

how transduction—the loss or addition of signification to a process—shapes a problem-oriented regime's material outcomes and encounters. Transduction and mixed semiotics are a way to account for the unreliability of software and the unpredictability of its implementation and delivery by elucidating what happens when, reciprocally and bidirectionally, diagrammatic processes are imposed on signifying interpretations and interactions. This, in turn, allows one to understanding why a sociotechnical regime's problem-orientation explains more about its relation to the world and itself, and why, specifically sourcery must be dispelled and why future media studies scholarship should incorporate thorough understandings of software engineering.

The significance of a Deleuze and Guattarian framework for media studies, of sociotechnical regimes to analyze and describe software engineering, allows a simplification of terms such that intersections between philosophical concepts—as proposed in media studies scholarship—and scientific functions can be analyzed in the contexts in which they operate. Just as the sciences should not attempt to redefine philosophical concepts, philosophical projects should not attempt to overcode scientific functions. Sociotechnical regimes offer a Deleuze and Guattarian framework for analyzing the intersections between a discipline, like software engineering, and the world in which it operates, produces, and mediates. The significant contribution this framework makes is to narrow the distance between media studies scholarship and the engineering practices and outcomes they purport to study. The gap is evident when evaluating current definitions of 'software,' 'programming,' and 'source code.'

This dissertation is an attempt to reboot parts of the media studies' scholarship associated with software. Using Deleuze and Guattarian methods to incorporate and elucidate the definitions and practices of software engineering borne of industry and STEM, the gap between practical definitions of software and philosophical ones can narrow in productive and surprising

المنارة للاستشارات
www.manaraa.com

ways. They believed that "philosophical concepts act no more in the constitution of scientific functions than do functions in the constitution of concepts," which means that the work of philosophy in evaluating scientific functions resides outside of a definitive, denotative domain (Deleuze & Guattari, 1994, p. 161). The issue with media studies scholarship about software is that it has unnecessarily attempted to redefine terms: Deleuze and Guattari viewed philosophical concepts as an event or virtuality that exists and affects beings whether they like it or not, but crucially recognized the actualization of things within a state of affairs as being governed by scientific propositions and functions. This distinction of philosophy from science, that the "actualization and counter-effectuation are not two segments of the same line but rather different lines" (p. 160), provides a basis for looking for intersections between a virtual concept and an actual function. So, rather than treating software as invisible, or fetishizing it, which leads to narratives of its existence and functions in our lives as mysterious, or of media studies' investigations into its nature which offer to 'pull back the veil' to only further redefine it, it may be more productive to consider it as always visible by working with its existing and science-oriented definitions. Tracing the connections between things—their intersections—describes their becomings without the need the redefine their terms.

On Software and Software Engineering, Programming, and Source Code

> No doubt states of affairs that are too dense are absorbed, counter-effectuated by the event, but we find only allusions to them on the plane of immanence and in the event. The two lines are therefore inseparable but independent, each complete in itself: it is like the envelopes of the two very different planes. Philosophy can speak of science only by allusion, and science can speak of philosophy only as of a cloud. If the two lines are

www.manaraa.com

> inseparable it is in their respective sufficiency, and philosophical concepts act no more in the constitution of scientific functions than do functions in the constitution of concepts. It is in their full maturity, and not in the process of their constitution, that concepts and functions necessarily intersect, each being created only by their specific means. (Deleuze & Guattari, 1994, p. 161)

Before diving into Deleuze and Guattari's (1994) epigraph to this section, we need to touch upon a connection in media studies scholarship that speaks to a theme of intersection between philosophy and science. Bernhard Siegert described media studies' scholarship as operating in a Kittlerian mode for the last twenty-years during the closing remarks of the 2018 Princeton Weimar Summer School for Media Studies. He stated that while it had worked well, the future of media studies required a broadening of the approaches used to study cultural techniques (incidentally, the research question for the Princeton Weimar 2019 session is, "What Happens When Practices Become Algorithmic Technologies?"). Siegert lamented that the study of cultural techniques must be connected to their practices. While Kittler may have learned to program, it is not clear that he understood the reasoning behind software engineering's axiomatics. Deleuze and Guattari offer a way past the Kittlerian mode for media studies scholarship. The divide between industry and STEM-based definitions of software, programming, and source code with media studies scholarship is stark, notwithstanding some exceptions. While there are several factors influencing this divide, including issues of education, training, and professional experience, Kittler's attitude about technology and software has shaped its definitions to the detriment of the field: software engineering is not as it is in industry or in society at large, but as it is through the lensing of Kittler's distrust and polemics.

المنارة للاستشارات
www.manaraa.com

While media studies' definitions for 'software,' 'programming,' and 'source code' incorporate aspects of industrial and STEM perspectives, more often the work attempts to redefine those terms, rather than *intersect* with them. Deleuze and Guattari's epigraph above is an argument for respecting the self-sufficiency of philosophical concepts and scientific functions: inquiry that relies on intersections between philosophical concepts and scientific functions are often not fruitful because they do not give each other the respect they deserve. Preceding Deleuze and Guattari's epigraph, philosophical concepts and scientific functions and propositions move in co-equal spheres: just as a concept requires one to return to "the event that gives its virtual consistency," it is also "necessary to come down to the actual state of affairs that provides the function with its reference" (Deleuze & Guattari, 1994, p. 159). Concepts cannot dominate functions, and conversely, functions cannot dominate concepts: events are "actualized or effectuated whenever [they are] inserted … into a state of affairs," but are "*counter-effectuated* whenever [they are] abstracted from states of affairs so as to isolate the concept." A concept, in fact, is treated as *amor fati*, e.g., "I was born to embody [the concept] as event because I was able to disembody it as state of affairs or lived situation." Concepts are there—and lived—whether one likes them or not, and should be accepted as such; but concepts, which are virtual, rely on the functions and propositions of science to become actualized in a state of affairs. While related, they are not equivalent, and attempting to equate them is a type of domination, or over-coding, that leads to confusion: "actualization and counter-effectuation are not two segments of the same line but rather different lines" (p. 160). It follows from Deleuze and Guattari's reasoning that, if concepts and functions are not the same, that they have independent trajectories as distinct lines, respecting their independence and self-sufficiency yields understandings about their wholeness and meaning within their respective virtual and

المنارة للاستشارات
www.manaraa.com

lived contexts. Such understanding ultimately frees inquiry looking at their intersections by eliminating the desire to dominate one or the other by redefining terms. Scientific thought-forms, Deleuze and Guattari argued, are self-sufficient within their own wholeness; the task of philosophical inquiry into the sciences is not to tell the sciences how their functions work, but to discover the intersections between virtual and actual, between the event of a concept as its consequences ripple throughout the lives of those enmeshed in a state of affairs.

The intersections Deleuze and Guattari argued for occur not in the formation of a concept or function, or even in its actualization, but in how they move and shape life in accordance with the concept of an event. Computer science and software engineering, like any scientific or engineering disciplines, are governed by ontological commitments and epistemologies that are distinct in their definitions, practices, and actualizations from the philosophical concepts' media studies investigates. An example of science respecting the self-sufficiency of concepts is found in Julio M. Ottino (2013) opinion piece written for the Robert R. McCormick School of Engineering and Applied Science at Northwestern University. In *View from the Intersection: Why We Need the Humanities and the Arts*, Ottino argued that the focus of the arts was not necessarily to solve problems, but to create questions, and that such a mindset allowed one to "[see] things in a completely new fashion," which is "ultimately what innovation is about." Humanistic inquiry and its methods, with their emphasis on concepts and the virtual, complement the work of engineers and scientists by allowing them to discover incongruities and question abstractions. The concepts of the virtual and the methods used for finding new questions were strengths to be leveraged, rather than over-coded.

As evidenced in the literature review here, media studies have often sought to over-code the functional and propositional definitions the sciences use. That scholars in media studies' have

المنارة للاستشارات
www.manaraa.com

argued that software be treated as a fetish—or treated it as a fetish purposely or unwittingly—is evidence of Deleuze and Guattari's concerns: the definitions of software from those who create it—e.g., computer scientists, software engineers, software developers, programmer analysts, programmers, and web designers—are considered only insofar as to dismiss and refute them, a practice contrary to the independence and self-sufficiency of concepts, functions, and propositions that Deleuze and Guattari argued for. For example, the industry and scientific definitions of 'software' has been over-coded and redefined by media studies' to such an extent that its wholeness and self-sufficiency as a distinct product from an independent thought-form has been ablated from its state of affairs. This means that the definition of 'software' in media studies is not the definition of 'software' from a scientific or industry perspective. If Deleuze and Guattari's argument for self-sufficiency is taken seriously, both fields would operate with co-equal definitions, spend time examining intersections of concepts and their manifestations in the actual without the need to redefine *already sufficient terms*. It simply is not the role of the philosopher, according to Deleuze and Guattari, to define how science works. Rather, it is far more productive to look for intersections between power or media effects and software and its constitution from the common definitions that respect philosophical and scientific denotations. If respected, the self-sufficiency of science and philosophy and art provide a baseline for cooperation by eliminating the impetus to over-code one another's methods and operations.

Over-coding is especially egregious in software's case, where the conceptual and axiomatic frameworks in which it is designed, implemented, tested, and maintained are often ignored or glossed over in favor of attempts to redefine what it is. For example, despite software being ubiquitous, it is only "allegedly" something that "[all] new media … rely on" (Chun, 2008, p. 300). Friedrich Kittler (1999) famously recognized that "[media] determine our situation,"

المنارة للاستشارات
www.manaraa.com

which is true (p. xxxix); he stated that “[inside] the computers themselves everything becomes a number,” which is true (p. 1); but he also claims that “[our] media systems merely distribute the words, noises, and images people can transmit and receive. … the only thing being computed is the transmission quality of the storage media,” which is hyperbole to the extent that it is false and easily disproved by investigating how Netflix maintains streaming quality under adverse internet transmission rates (p. 2). Interactions with ‘software’ are always ‘computing’; an idling modern computer is always servicing its many processes, checking for user input events and network events. The software that humans interact with is always running, and therefore always visible to an important extent.

Chun (2011) states that definitions of software are often reduced to a kind of “[visible invisibility],” because “[software] seems to allow one to grasp the entire elephant because it is the invisible whole that generates the sensuous parts” (p. 300). Visibility does not always equate to ‘scrutibility,’ to play with the word: software presents only as much of itself as it needs to through well-defined interfaces, a topic which Alexander Galloway (2012) wrote about at length. So as a matter of perspective, software is visibly invisible, because aspects of its implementation are hidden, stored in memory cells, executed by processors, converted to electron flows; but as a matter of perspective, it is important to note that such a description describes the consequence of an effect: users press a button or swipe next on their smartphone’s touch-screen. From the ubiquity of the smartphone device to the integration of computational interfaces in daily life—like desktop or laptop computers, tablets, ATM machines, point of sale (POS) devices in stores, gas stations—it is safe to say that a definition of software exists that resists Chun’s allegation about it: software already has a definition and has reached a “full maturity” that not only no longer needs “the process of [its] constitution” to be explained, and it is therefore

المنارة للاستشارات
www.manaraa.com

inappropriate—from a Deleuze and Guattarian perspective, at least—for philosophical epistemes to attempt to redefine it (Deleuze & Guattari, 1994, p. 161). Media studies' scholarship will only benefit from integrating the definitions of established scientific and engineering functions and propositions, because often the consequences of their effects are more important than the details of their implementation.

While much effort has been spent on attempting to define the materiality of code and the software it becomes, many important works in media studies' have ignored the *how* and *why* of code, the practices leading to its creation, and the communicative and human practices which are arguably more important than the code itself. Evidenced by Zara Dinnen's (2013) explanation that notable scholars like Katherine Hayles, Johanna Drucker, Matthey G. Kirschenbaum, Wendy Chun, Alexander R. Galloway, Adrian Mackenzie, and Eugene Thacker have "a shared concerned with code as language and code as text" (pp. 175-176), media studies scholars have spent most of their time examining the nature of source code and the activity of programming. Yet, while one or more derivations of terms like "program," "programming," "programmer," "code," "coding," and "source code" are indexed topics in the appendices of the major works of N. Katherine Hayles (1999, 2005), Alexander R. Galloway (2012), Lev Manovich (2013), Wendy Hui Kyong Chun (2008, 2011, 2016), Rob Kitchen and Martin Dodge (2011), and Friedrich Kittler (2008; 2010; with Gumrecht, 2013), terms like "software" in conjunction with "engineering," "development," or "developer" are absent. "Programming" and "programmer" are an action and identity that turns an individual into an analogous author writing a text. By excluding the broad sets of patterns and practices software engineering uses to design, develop, test, document, and maintain software, media studies have overemphasized the production of 'source code' to the extent that 'programming' is now synonymous with software development.

المنارة للاستشارات
www.manaraa.com

Respecting the self-sufficiency of a term like ‘programmer’ or ‘source code’ would reduce the error of conflating those terms with software engineering. By redefining the terms, media studies scholarship has limited itself to a simplified definition of software that fetishizes ‘programming’ at the expense of a nuanced and sophisticated understanding of the communicative processes through which problems become solutions, and solutions become software, while allowing for their self-sufficiency and wholeness is the way forward to introduce such nuance into media studies’ scholarship.

Current media studies and scholarly definitions of software range in degrees from assertions that it does not exist (Kittler, 2013, p. 223), to omitting its definition in theoretical discussions (Hayles, 2005; 1999). As pointed out earlier, Chun (2011) argued to “make our computers more productively spectral” by allowing scholars to accept the “the unexpected possibilities of source code” by fetishizing it (p. 20); while fetishization effectively “allows one to visualize what is unknown” by “[substituting] images for causes,” those images are flawed by being, at best, oversimplifications, and at worst, technically and *socially* inaccurate. Examining code for what it is—by respecting the self-sufficiency of its definition in Computer Science, and software engineering—allows it to be accurately described, explained, and understood within the actualized contexts in which it is produced. Fetishism is and has been the wrong move for media studies’ investigations of software, programming, and source code because it has led to a disassociation from the technical and social contexts in which they exist, are practiced, and are enacted. Contrasting definitions of ‘software’ in the contexts in which it is created and by those creating it to media studies’ perspectives provides a sense of how Chun’s argument for fetishism—and the fetishism’s presence in popular theoretical definitions and discussions about software (c.f. Winthrop-Young, “Hardware/Software/Wetware” in Mitchell & Hansen, 2010)—

المنارة للاستشارات
www.manaraa.com

undermines scholarly investigations of software mediation by attempting to *define* it as something it is not. The issue at stake here is not with media studies scholarship that *describes* it, from the perspective of the intersections it makes with the concepts of our lives, but with scholarship that attempts to *re-define* it, which is contrary to Deleuze and Guattari's (1994) argument for the self-sufficiency of functions and concepts. This literature review, although brief, provides evidence for the extent to which 'software' as a technology is fetishized in media studies. Additionally, as this review contrasts the industry and media studies definitions of software, programmer, and source code, it will illustrate how productive a study of software engineering can be for future media studies' scholarship if the Deleuze and Guattarian perspective about the self-sufficient domains of functions and concepts in science and philosophy are observed.

From the perspective of Computer Science and software engineering, software comprises the communities of practice that produce the instructions which compel a computer to perform work. The word 'software' began to be used during the late 1950s by the engineers and scientists of the time to describe the nature of the operating contexts of a "stored-program computer" (Cambell-Kelly et. al, 2014, p. 168). Paul Niquette (1995) claims to have "coined the word 'software'" in 1953. From the perspective of a scientific and practice-driven thought-form, software was defined as the "set of instructions that cause [a] computer to operate; a program or set of programs" (Kohanski, 2000, p. 226). It's common definition, according to Oxford Dictionaries, includes "the programs and other operating information used by a computer" to perform work with the hardware upon which it executes. Dictionary.com defines software in relation to computers as "the programs used to direct the operation of a computer, as well as documentation giving instructions on how to use them." In Harvey M. Deitel and Paul J.

المنارة للاستشارات
www.manaraa.com

Deitel's (1994) classic book on C programming, *C: how to program*, software is defined as "the instructions you write to command the computer to perform actions and make decisions" that allows hardware to operate (p. 3). The instructional perspective—that of guiding the work computers perform—is echoed in Paul Cerruzi's (2012) MIT Press effort, *Computing*, where he describes the tasks and guiding principles of software and hardware design as managing "the complexity from the lower levels of logical circuits to ever-higher levels that nest above one another" so that a computer system can perform meaningful work (p. 83). While a duality may have existed between hardware and software at the inception of mechanical computation, the reality now is that software is often the reason for new hardware developments, because "[every] machine requires a set of procedures to get it to do what it was built to do," and "[only] computers elevate those procedures to a status equal to that of the hardware" (p. 56). Software enables hardware which enables software, and so on in a process of recursion. Computer scientists typically view computers, historian Nathan Ensmenger (2010) explains, as "simply a device that can run a certain kind of software program"; it does not matter what kind of device it is if "it is programmable" (p. 5). His most succinct definition of software states that it "is what makes a computer useful," and without it a "computer is … irrelevant, like an automobile without gasoline or a television set without a signal." Computer hardware requires software, which is a set of instructions that cause the hardware to work on a specific task. Extending Ensmenger's gasoline metaphor a step further allows software to be placed in the contexts in which it is drilled for, refined, stored, and shipped: Michael Mahoney (2008) argued that producing a history of software is hard, because all "current" software is in fact "'legacy'" software, because of the way "the models and tools that constitute" it "reflect the histories of the communities that created them and cannot be understood without knowledge of those histories,

المنارة للاستشارات

www.manaraa.com

which extend beyond computers and computing to encompass the full range of human activities" (p. 8). Software encompasses the knowledges and processes that lead to the production and distribution of instructions that impel any kind of mechanical or electronic device to perform work of some kind, so knowing what software is requires knowledge of how software is made.

In media studies, software is defined as largely unknowable and un-seeable, as instructions derived from source code, and that computers—and therefore software—created themselves. While some definitions of software tend to half-align with those found in Computer Science and software engineering, like Rob Kitchen and Martin Dodge's (2011) which states that "software consists of lines of code—instructions and algorithms that, when combined and supplied with appropriate input, produces routines and programs capable of complex digital functions" (p. 3), they tend to exclude the communities of practice that design and produce it. Software tends to be defined as instructions, and depending on the scholar, may not exist at all if it did not need to intersect with humanity: Kittler (2010) has described computer instructions in terms of electrical voltages (p. 226), which he elaborated on in his essay, "There Is No Software" (2013), by reasoning that "elementary code operations, notwithstanding their metaphorical promises (e.g., 'call' or 'return'), amount to strictly local manipulations of signs and therefore (more's the pity, Lacan) to signifiers of varying electric potentials" (p. 223). He ultimately argued that software would not exist at all if computers "did not need … to coexist with an environment of everyday languages." Or software exists, but "is extremely difficult to comprehend," which Wendy Chun (2008) attempted to simplify by treating it as a metaphor that "illuminates an unknown" which "does so through an unknowable (software)" (p. 2). Software, as such, has a degree of inscrutability: computers are "mediums of power" and software is a kind of "vapory materialization" with a "ghostly interface." In *Software Takes Command*, Lev

المنارة للاستشارات
www.manaraa.com

Manovich (2013) explains that the languages of modern society are the languages of software, and that software "is the invisible glue that ties" all aspects of modern societies "together" (p. 8). In his essay, "Hardware / Software / Wetware," Geoffrey Winthrop-Young (2010) dismisses the Oxford English Dictionary's definition of 'software' as "pithy" because the "computer / software binary restages the old boundary disputes between body and spirit," which makes an easy definition of the term complicated (p. 189). He echoes the 'unknowable' and 'un-seeable' definition by explaining that it may be only "the most visible part of a bewildering edifice … that for the vast majority of users remain out of sight," which in turn "interposes itself between the user and the basic operation of the servile tool" (pp. 190-191). Winthrop Young cites Kittler's assertion in "There Is No Software" that while all software operations "can be reduced to hardware operations" (p. 193), computers are "both very social and irredeemably autistic," to an extent such that "it is not enough to say that human communication is modeled on the computer" (p. 197), implying that computers are endowed with a kind of self-generating agency, independent of the humans who engineered and produced them. N. Katherine Hayles (1999), for example, used an assertion by Alan Turing to empower hardware with a sort of disembodied intelligence and post-human agency that is proved through the 'Turing test,' which Alan Turing explained as one where, if by 'talking' to a computer, a human could not determine if they were talking to a person or a program, the computer was thinking (pp. xi-xii). As used by Hayles, Turing's test ignores the software techniques and mathematical advances necessary to encapsulate grammar parsing and content analysis required to even begin understanding the problems associated with accurately understanding the meanings behind human language. Ultimately, Winthrop-Young's (2011) explanation of Kittler's attitude toward software informs media studies general transgression of it: "what [irked] Kittler is that … user-friendly *software*

المنارة للاستشارات
www.manaraa.com

tricks us into believing that we are in charge of the computer. We continue to think of computers as mere tools, and this, in turn, serves to perpetuate our narcissistic self-image as *homo faber*, man the toolmaker" (p. 75). Software is untrustworthy, inscrutable, and aligned against humanity. So, despite Manovich (2013) recognizing that "software as a theoretical category is still invisible to most academics, artists, and cultural professionals interested in IT and its cultural and social effects" (p. 9), there has been an explicit rejection of the definition of 'software' as it has been employed by the thought-forms responsible for its maintenance in favor of a broad cynicism.

From an industry perspective, programmers (synonymously referred to as a developer, software developer, or software engineer) are the people who design and develop software, and 'programmer' is a title encapsulating practices beyond those of writing computer code. According to PC Magazine's encyclopedia of terms, the "software business is a service industry that involves human thinking almost exclusively," which "contrasts with computer hardware or any other industry that makes equipment, whereby manufacturing is a major part of the business." The Oxford Dictionaries definition states that a programmer is "a person who writes computer programs," which is refined in Kohanski's (2000) definition, which explains that a programmer is a person "trained in one or more computer languages" who uses those languages to create computer programs (p. 224). The Bureau of Labor Statistics (BLS) of the United States Department of Labor (2019) has entries for computer programmers and software developers: programmers "write and test code that allows computer applications and software programs to function properly," while software developers "are the creative minds behind computer programs," definitions which flow into one another. Software developers, systems analysts, and other areas of expertise working with software have tended to be "lumped together by outsiders

المنارة للاستشارات
www.manaraa.com

as programmers" (Ensmenger, 2010, p. 14). The term "programmer" arose from "coder," which "implied manual labor, and mechanical translation or rote transcription," a mistake John von Neumman and Herman Goldstine—authors of the first textbook on programming—quickly realized as erroneous: "[what] had been expected to be a straightforward process of coding an algorithm turned out to involve many layers of analysis, planning, testing, and debugging" (p. 15).

Programming and source code are covariant terms. The job of programming computers, or the task of 'programming problems' as Cambell-Kelly et al. (2014) called it, grew to encompass "more than simply writing efficient and bug-free code": as the complexity of software increased, the individuals designing and writing the software became increasingly involved "in a broad range of activities that included analysis, design, evaluation, and communication—none of which were activities that could be easily automated" (p. 183). Today, job titles and responsibilities tend to refer to a type of expertise relative to the contexts in which a problem resides, such as those outlined by a corporation's needs, or a societal issue. For example, for one company, a web developer's job responsibilities might encompass expertise in a range of skills that overlap with 'software engineers' in another company: titles can be highly interchangeable. While seniority and autonomy tend to determine the extent to which developers are required to cultivate many skills, "full stack" developer positions are increasingly common. Trista Liu (2017) describes full-stack developers as having wide "horizontal [skill trees]" who have knowledge at width, rather than at depth. A "full-stack" developer is expected to meet with clients, gather requirements, produce design documents, develop the software, test it in a staging environment, gather feedback from the clients prior to releasing the software into a production environment, refine it, and then maintain it over time. Despite the caveats of the role—the

المنارة للاستشارات
www.manaraa.com

emphasis on width rather than depth of a skill set—most of the tasks previously described are common to any Software Development Life Cycle (i.e., Waterfall, Agile, etc.), and rely as much on communication skills as purely technical ones. Clear communication of and about a problem mitigates errors by increasing the likelihood that the developed software reflects customer requirements; programming is but one area of expertise software developers must gain competency in.

Definitions of programmers and programming, in media studies, vary in degrees from industry positions. Manovich (2013) hews closest to established definitions, describing programmers as those who have “programming skills,” with “access to a computer, a programming language, and a compiler” (p. 93). Programming skills produce software. He explains that “software development is an industry” which is “constantly balancing between stability and innovation, standardization and exploration of new possibilities.” Kitchin and Dodge (2011) also work from industry positions, describing programming as a skill, the purpose of which is to “construct a set of coded instructions that a microprocessor can unambiguously interpret and perform in ongoing flows of operations” (p. 25). Programmers produce code which are ‘unambiguous’ instructions for a computer to follow. The code that programmers produce should be clear enough for a computer to follow, and for a human to understand, maintain, and reproduce in other contexts, for other problems.

One of the most influential definitions of ‘programming,’ programmer, and code—the issues entangled with the skills and acts of producing software originates with Kittler, which has introduced a stark divide in terms of the ambiguities (or lack thereof) of source code and what it is which programmers produce. His definition of ‘programming’ is commonly reflected in media studies scholarship, which commonly describes it as an analog for writing, efforts of which have

www.manaraa.com

led to attempts to redefine it in terms of signification (rather than asignification), e.g., ambiguity in favor of unambiguity. For example, in a recent collection of essays about Kittler studies published in 2015, *Kittler Now: Current Perspectives in Kittler Studies*, Stephen Sale summarized Kittler's perspective on *programming languages* as those which have "toppled the monopoly of ordinary language" and are "multiplying in order to obfuscate the technical decisions made at a hardware level" (p. 64). Kittler (2013) set the mold for this interpretation of computation and programming when he conflated writing with a "peculiar kind of software … at the incurable confusion between use and reference" (p. 220). But if what Winthrop-Young (2011) pointed out about Kittler's attitude about software is true, that "he came to deny its very existence" because "it can be reduced to basic hardware operations," programming cannot exist either (p. 75), then what kind of writing is programming? Kittler believed that "[we] must study basic programming and operating languages in order to overcome their dependence on the software opium handed out by the industry" (p. 77), so software—in its non-existence—must be a form of power or coercion, which is a theme Chun (2016) argued in later work: computers are a medium that are "essential to organizing and managing, assessing and predicting—that is, programming—populations and individuals" (p. 19). Programming then becomes a kind of writing that inscribes command and control logics onto the canvas souls of human beings. So, programming, as a kind of writing, produces text that may not exist at all, but at the very least produces something that is a form of power, which is itself a form of fetishism for the mechanisms, processes, and social and machinic relations that categorize and coerce humans.

'Power' and this kind of 'command and control' narrative is evident when Chun (2011) recognizes programming as an axiomatic but describes it in terms of writing. Despite using Deleuze and Guattari's reasoning, she questions rather than respects the industrial definitions of

المنارة للاستشارات
www.manaraa.com

programming by describing commands issued to a computer using a programming language as a "certain logic of cause of effect, a causal pleasure that erases execution and reduces programming to an act of writing" (p. 101). Here writing implies a hermeneutic framing, which are the domain of concepts, in terms of Deleuze and Guattari; source code, defined from an industry and scientific perspective, is less ambiguous. Chun had, earlier in her work, defined programming as a sort of logos which "turns *program* into a noun—it turns process in time into process in (text) space" (p. 19), the emphasis on 'text' is important, because it indicates the presence of a bias in the interpretation of another thought-form's definitions, attempting to coerce in this case something functional and propositional into that which is signifying and open to hermeneutical interpretations. From an industry perspective, programming is writing insofar as it is 'writing' instructions a computer will follow that humans can understand for further development purposes. Kohanski (2000) defined source code as "a program in the form of statements written in some language that a human being can understand. A compiler or assembler will convert source code to object form, and the linker will combine one or more objects into an executable program" (p. 226). What could media studies gain if it moved away from Kittler's distrust of software toward thorough understandings and appreciations of software development's axiomatics and problem-orientations? At this stage, regardless of whether a machine learning, artificial intelligence, or human intellect is generating 'programming,' each are generating instructions for a machine. Within the machine domain, source code is what it is, in a cloyingly tautological sense; it is not until software intersects within human society that it begins to take on new meanings. Arguably, software always mediates, but not from a position of ethereal vapor, or electrical haze. The dominant definition of software rightly resides in industry;

المنارة للاستشارات
www.manaraa.com

making sense of software requires that the practices leading to its production are enumerated and understood so that narratives of sourcery can be dispelled productively.

There is an impasse in the ways philosophy and science speak of each other. In the epigraph to this section, Deleuze and Guattari (1994) stated that "No doubt states of affairs that are too dense are absorbed, counter-effectuated by the event, but we find only allusions to them on the plane of immanence and in the event" (p. 161). This means that, for however a state of affairs, in its concrete, actualized form is configured, it alludes to philosophical concepts through indirect means, and vice versa: philosophical concepts will "counter-effect[uate]" an event in response to a change in a state of affairs, but do not directly affect the composition of the functions and propositions that configured it. Philosophical concepts and scientific functions relate, rather than correspond, with each other: they are self-sufficient. This is important, because as Deleuze and Guattari argued, "[it] is in their full maturity, and not in the process of their constitution, that concepts and functions necessarily intersect"; the task of philosophy is not to redefine the functions of the sciences, nor is it the job of the sciences to redefine concepts emerging from philosophy.

Software, programming, and source code represent an impasse where the key terms provided by science are modified in such a way that their fundamental meanings are undermined by philosophical overcodings. This represents a shift away from an actual state of affairs into explorations of 'intersections' that reside primarily in a philosophical domain. As evidenced by the comparison of industry and media studies' definitions of 'programmer,' 'source code,' and 'software,' their self-sufficiency has not been respected. Deleuze and Guattari's thought-forms were argued for to refocus attention on intersections, on the becomings of relations between concepts and functions, so that new things can be discovered, described, defined, and offered

المنارة للاستشارات
www.manaraa.com

forth in such a way as to give rise to new concepts and reflective functions, “each being created only by their specific means” (1994, p. 161). Self-sufficiency is a way to provide a basis for collaboration, hence the purpose of this dissertation project is to model Deleuze and Guattari’s argument at a point of intersection within the scholarship and history of software engineering.

The Visibility of Software Requires Software Engineering

Treating software as always visible requires that software engineering be defined in terms of what it is not, which is Computer Science. It further requires that the practices and reasoning of software engineering be both enumerated and respected for the purposes they originated, to solve problems computationally. Software is the visible expression of a solution to a problem. Its visibility not only outlines its functions but provides insights into the ways it was designed and implemented. Understanding software from the perspective of software engineering practices breaks the ‘invisible’ narrative apart by revealing the common scenarios and axioms that lead to its development and deployment under most circumstances. Software became ubiquitous as soon as it was deployed broadly in society at large within the cheap read only memory (ROM) chips, embedded processors, and integrated circuit boards of toys and consumer products in the late 1970s and early 1980s; it is not mysterious if subjected to scrutiny that takes the *processes* used to problematize its solutions into account. Treating software as invisible, or as a fetish, is simply a failure to engage with and understand the broader communities of practice that produce it. Software is the visible artefact of a problematic; software is the visible solution to a problem.

Sociotechnical regimes form around a problematic. For understanding software from an engineering perspective, what matters most is how a problem is translated from one domain, that may be continuous and analog rather than discrete and digital, into a computational one: can the

المنارة للاستشارات
www.manaraa.com

problem be solved computationally? If so, what should the solution look like? What is the solution supposed to do? Can the problem be broken down into smaller parts, or must it be solved in its entirety? Software and its effects are always visible from the perspective of an engineer because matters of implementation allude to issues of design, which is always rooted in a problematic, which Deleuze (1994) defines as "the ensemble of the problem and its conditions" (p. 177). Problems, for Deleuze, are immanent to their conditions: "the complete determination of a problem is inseparable from the existence, the number and the distribution of the determinant points *which precisely provide its conditions*." Solutions, on the other hand, tend to "conceal the problem," because they become representational and experiential, factors which tend to outweigh and overshadow the broader implications a problem may have (p. 178). If software is treated as a solution, its implementation and manifestation is less important than the problem it seeks to produce a "solution-instance" for (p. 178); rather, an instance of 'software' or the manifold of processes in a 'software environment,' from a user's desktop operating system (OS) to those run in server farms by Microsoft or Google provides clues to an orientation that lead to its implementation and purposing.

The sociotechnical regime, developed here over three chapters, incorporates Deleuze and Guattari's respect for the distinct nature of scientific functions and philosophical concepts. By defining what a sociotechnical regime is, relating software engineering's history in terms of a regime, and describing how that regime intersects with society at large through processes of transduction based on signifying and asignifying transmissions, this project offers an alternative to the Kittlerian perspective on software which lead to Chun's later argument for its fetishization. Software is always visible, even when its source code is inscrutably inscribed in the ROM chips of a cheap toy, if it is considered as evidence of a broader problematic, itself belying a regime

المنارة للاستشارات
www.manaraa.com

incorporated to solve problems within and for those conditions. Understanding software engineering practices can expose a broad array of mediating effects that work within and without the industrial practices producing the software that mediates consumers, users.

المنارة للاستشارات

www.manaraa.com

Chapter 2: Problem-Oriented Sociotechnical Regimes

Problems work through their conditions. Truth, for a software developer, flows from the contexts in which they find themselves into the tasks they must complete. And for software, the problems and conditions of its encounters with us, in our situations, to borrow the spirit of Kittler's (1999) language, are mediated by 'software engineering' processes and identities long before the software itself is produced and deployed to act upon us. Software engineering processes encompass the machine and the software; while it is obvious that 'engineering' implies a way of solving problems, the fetishization evident in media studies indicates that problems are being separated from their conditions. Kittler's (2013) attitude toward software and hardware has greatly influenced media studies, and while the assertion he made about the Central Processing Units (CPUs) in machines is true, in that they "can do both less and more than [their] data sheets" reveal (p. 216), processors and their data sheets have not been placed in the broader software engineering contexts that recursively generate them. In fact, what is most striking in media studies is that, while the spirit of Kittler's suspicion is borne out in the works of Alexander Galloway (2012), Mark Sample (2013), or Felicitas Kraemer and Kees Overveld (2010), seemingly lost in scholarship examining the technics of computation is one of the most striking points he made in his famous essay, "Protected Mode," where he stated that "[the] lovely phrase 'source code' names the literal truth" (p. 218). De-fetishizing source code requires that the communicative practices that feed into its creation be factored into examinations of software, into the practices which cohered as a sociotechnical regime called software engineering. Taking into consideration what a regime is allows those data sheets to be viewed in the context of the software engineering practices that recursively fed them, and fed from them, which generated them, which produced the CPU model they detail. Such considerations will allow media studies

المنارة للاستشارات
www.manaraa.com

to move away from the nebulous opacity Kittler thought the codes in CPUs represented, from a principally fetishistic perspective of source code and programmers, to one that factors in the all-to-human considerations shaping the truth of a software developer, which is that they are always enmeshed in the problematics of their situations. Kittler's statement about 'source code' as literal truth should be taken literally, because it represents a perspective Deleuze and Guattari (1994) argued for by stating a self-sufficient definition of code that does not attempt to overcode or deterritorialize: "philosophical concepts act no more in the constitution of scientific functions than do functions in the constitution of concepts" (p. 161). Source code, for a computer, is literal truth, having been "created only by [its] specific means." This chapter develops the concepts of problem-orientation and the sociotechnical regime and describes a Deleuze and Guattarian framework for understanding how problems organize and coalesce into practices and identities. Those practices and identities are captured by sociotechnical regimes, which are sets of technical and social practices forming relatively stable recursive values (operating principles, statements of purpose, mores), identities (the project manager, the computer scientist, the programmer analyst), and product outcomes (software, user stories, unit tests, memos, code) as means to solve problematics.

This chapter establishes a basis for thinking about software engineering as a problem-oriented sociotechnical regime by looking at three areas. First, it starts by examining problematics, considered by Deleuze to be a combination of a problem with its conditions, which looks at its Bergsonian origins, the significance of problematization to Deleuze and Guattari's (1977) *Anti-Oedipus*, its role in the conceptual and analytical approaches to *What Is Philosophy?* (1994), and why problematization is important for understanding their thinking about the sciences. Next, it continues by describing how problem orientations emerge from a problem's

www.manaraa.com

material and experiential encounters with the world, which provides a basis for understanding how problems can lead to the organization of a discipline focused on solving them, leading to disciplines encapsulating identities and practices, like Computer Science. Finally, it defines sociotechnical regimes as a problem's manifestation in a human world, developing an analytical framework using Deleuze and Guattarian concepts to account for and describe why a regime's development of social and technical practices coalescences around continuing encounters with its governing problematics. It concludes by describing why problem-orientation and sociotechnical regimes matter for understanding software by mapping software engineering as a sociotechnical regime and briefly describing three of the fundamental problems guiding and complicating its organization and focus as a regime, a history of which is developed further in Chapter 3. The work outlined here provides a framework for understanding the relationships between problematics and sociotechnical regimes and is the first of three steps this project makes in responding to its exigence. For media studies, the concepts of problem-orientation and sociotechnical regimes and their relationships are the basis for a means to de-fetishize (Chun, 2011; 2008) and disentangle the identity of the 'programmer' with the primacy of 'source code.' The problem-oriented sociotechnical regime encapsulates a nuanced understanding of how the broad array of social and machinic processes used to design and implement software mediate problems themselves, which, by altering the problems they seek to solve in subtle and explicit ways, have greater consequences in a software-dominated world than source code's materiality or treating the programmer as its soul author.

Before proceeding into this chapter, it is worth unpacking 'problem-orientation' and 'sociotechnical regime' in general terms, to frame and reflect on the role this chapter plays in responding to part of this dissertation's exigence. From there, I will briefly relate my definitions

المنارة للاستشارات

www.manaraa.com

to Deleuze and Guattari's theoretical considerations. Framed as such, these cursory definitions will provide a conceptual basis for relating the dense and nuanced works of Deleuze and Guattari to the overall shape of a regime like software engineering. To begin with, a problem-oriented sociotechnical regime like software engineering enacts many types of processes to complete the work necessary to solve problems. I define problem-orientation as a type of task-based, process-driven form of effort that parses problematics according to some orienting principles that dictate, by and large, how the problem in a problematic is to be understood and solved relative to its conditions. I employ Deleuze's (1994) definition of problematics, which is the combination of a problem with its conditions, which has several implications: first, problems are inseparable from their conditions; second, problems reside in a set of conditions; and third, problems arise from their conditions (p. 177). According to the Oxford English Dictionary, a process is, in its noun-form, "a series of actions or steps taken in order to achieve a particular end," which for computing is "a series of interdependent operations carried out by computer"; as a verb, process becomes the performance of "a series of … operations on (something) in order to change or preserve it," and for a computer, 'to process' becomes how a computer "operate[s] on (computer data)" using a program. Problem-orientation is a way of describing the types of tasks software engineers perform from a Deleuzian perspective. For software engineering, the terms that fall under the Software Development Life Cycle (SDLC) and Software Testing Life Cycle (STLC) processes, like structured programming, waterfall, or agile, are considered methodologies that govern how problems are interpreted relative to their conditions, how work is conducted according to those conditions, and what constitutes a solution for those conditions. Problem-orientation describes a mode of ideation and being that governs the processes used to instigate becomings, e.g., how software engineers produce software.

المنارة للاستشارات
www.manaraa.com

On the other side of ‘problem-oriented,’ I define a sociotechnical regime as a collection of practices and knowledges and neighboring relations by which disciplines distinguish and perpetuate themselves. This concept is born out of necessity, because a large part of what I believe leads source code and programming to be overemphasized in media studies is rooted in the conflation of ‘software engineering’ and ‘software engineers’ with contemporary understandings of ‘Computer Science’ (that inevitably refer to Alan Turing and computation in the universal, abstract sense). Basically, if software engineering is not seen as distinct from Computer Science, the ideation of universal computation and programming as writing another form of text will always favor understandings of ‘programming’ as acts of problem-solving distinct from conditions. This leads to arguments in media studies that avoid the broader implications of software by ignoring the most important parts of its development, how its developers negotiate, understand, and overcome the conditions in which (and for which) it is developed. I use Deleuze and Guattarian concepts to not only parse *related* science-based disciplinary identities, but to show how those identities can employ similar processes—like programming—for not only different purposes but weighted with different values. De-fetishizing source code and programming requires that software engineering be recognized as a distinct, but related, discipline to Computer Science while emphasizing the significance of process over singular aspect of implementation, of actual process over abstract universalism. Sociotechnical regimes represent a framework employing Deleuze and Guattari’s concepts of the event, the plateau, the rhizome, smooth and striated spaces, and the incorporation of and relation to a plane of reference to understand how a software engineering can become a discipline distinct from Computer Science, even as it produces (and reproduces) the functions and propositions related to computation.

المنارة للاستشارات
www.manaraa.com

Problematics

This section defines problematics and discusses issues pertaining to their discovery and description in the sciences—a topic Deleuze and Guattari (1994) explored in their last collaboration—and applies it briefly to computational theory and practice. The purpose of this section is to explain what problematics are, and why it is difficult to separate problems from their conditions. This section describes and defines problematics in relation to a common media studies narrative, e.g., the abstract idealized form of computation produced by Alan Turing's oft-quoted concept of the universal computer, to show why and how *solutions* are often mistaken for their *problems*. Deleuze (1994) used the term 'problematic' to describe the inseparability of a problem from its conditions; problematics serve as the conceptual basis for teasing the problem of computation away from its solution ('universal machine'), and in so doing will root the actions and products of programming in a larger set of actualities and processes. This reconsideration of problems and conditions will act as the basis for seeing through the fetishism of the 'abstract universal' narrative and the conflation of programmer and source code with software. So, while software may be vaporous or cloudy in the sense Chun (2008) invokes, in that it runs everywhere and has a complex form of materiality (electrical signal, compiled executable, or source code), this section establishes a basis for understanding the significance of problematics for theories of software. Bound as such within a broader problematic, software's materiality transcends its programming, compilation, and execution; software is rather the material of 'becoming,' of investments of the processes that bring it to life, those of requirements gathering, specifications, design, programming, testing, quality assurance, documentation, and maintenance. Because of problematics, modern software—up until the point its development is abandoned—can be understood as a materiality of continuous delivery, of the continual

المنارة للاستشارات
www.manaraa.com

mediation of problems by conditions and vice versa. Understanding problematics begins to bind the concepts of software, programming, and source code to the practices which bring them to life. In so doing, this section sets the stage for the next section, 'Problem-Orientation,' which details how actions propositionally bind to practices.

Problematics, defined by Deleuze (1994) as the combination of a problem with its conditions, demonstrates a form of materialism that is important for understanding problem-orientation (p. 177). Deleuze's understand of problematics helps to both understand the allure of the abstract universal computer narrative, and to diverge from it. Problems are defined by Oxford Dictionaries in three ways that are relevant here: as "a thing that is difficult to achieve or accomplish"; as "an inquiry starting from given conditions to investigate or demonstrate a fact, result, or law"; and as "a proposition in which something has to be constructed." According to Todd May (2005), Deleuze believed that problems are "an open field in which a variety of solutions … take place," and that "[it] is the problems rather than the solutions that are primary" (p. 84). Of those definitions, the terms "thing" and "proposition," and phrase "inquiry starting from given conditions" are significant, because they demonstrate Deleuze's thinking by demonstrating how problems are things, they are described in terms of propositions, and they are rooted in a set of conditions. Deleuze argued for a definition of 'problem' that disconnected it from 'solution': "Identities come later, as particular solutions to the problems that being places before us, the problem that being is. To confuse those identities with being is to confuse the actual with the virtual. It is to confuse solutions with problems" (May, 2005, p. 85). To avoid confusing solutions with problems, Deleuze (1994) argued that we must reject the notion that "problems are given ready-made, and that they disappear in the responses or the solution," but actively probe for problems in the conditions in which they arise (p. 158). Problems have "their

المنارة للاستشارات
www.manaraa.com

own sufficiency" (p. 159). Problems determine what *is* possible by linking propositions to sets of conditions. They are not transcendent in the Kantian sense, but distinctly material, immanent in that they emerge from their conditions.

This is where a problematic can parse an abstract notion of computation like Alan Turing's (1936) to distinguish the problem of 'general computability' from the solution of a 'universal computer.' The concept of the universal computer, called a Universal Turing machine, is that to complete *any* kind of work with a computer, the computer must have a set of self-describing and generalized instructions that defined the way it worked on the data associated with the problems any human could describe. Essentially, a Universal Turing machine is a "single machine which can be used to compute any computable sequence" (p. 241), because its programming (provided by an infinite reel of tape) not only described the data to work on but defined how the instructions worked for each operation using that data. This explanation represents a solution to the problem of generalized computation because it cannot exist without describing machinic assemblages: "The machine is supplied with a 'tape' (the analogue of paper) running through it, and divided into sections (called 'squares') each capable of bearing a 'symbol'" (p. 231). Considered another way, the problem of general computability and of making machines perform any kind of work cannot be extracted from its machinic conditions.

While it seems obvious that computers imply machinic actualizations, that Turing machines cannot be imagined without using a metaphorical tape reader indicates the extent to which problems are seen in terms of their solutions. Thomas Osborne (2003) uses the term 'problematology' to describe the of work of studying problems, offering a way forward in terms of distinguishing problems from solutions while accounting for the influence conditions have on their problems. He traces the influence of problem-orientation in Foucault and Deleuze's work

المنارة للاستشارات
www.manaraa.com

by examining the role of Canguilhem and Bergson, respectively. While Foucault is not the emphasis of the work conducted by this dissertation, Osborne importantly points to how a kind of problem-orientation resulted in Foucault's important works. Canguilhem believed that problems, rather than theories, were a "way of giving substance to the history of thought" (p. 3), and his project, outlined in The Normal and the Pathological written in the 1940s, influenced Foucault's historical epistemologies, using genealogical methods. As Koopman (2013) states, "[genealogies] articulate problems. But not just any problems. Genealogies do not, for instance, take up those problems that come with supposed solutions readily apparent" (p. 1); Foucault's discontinuous histories, criticized by his historian contemporaries, are discontinuous precisely because they must be to articulate a network of relations around which a problem germinates. The problems of life are almost never linear, but they are progressions over time, and are difficult to imagine without referring to their conditions.

Similarly, Bergson's emphasis on problematics influenced Deleuze's work to the extent that Bergson's beliefs about problems emerge (pointedly) in Deleuze's work in *Difference & Repetition*, *Logic of Sense*, and *Foucault* (p. 7). Bergson argued that

> [the] truth is that in philosophy and even elsewhere it is a question of finding the problem and consequently of positing it, even more than of solving it. For a speculative problem is solved as soon as it is properly stated. By that I mean that its solution exists then, although it may remain hidden and, so to speak, covered up: the only thing left to do is to uncover it. But stating the problem is not simply uncovering, it is inventing. (Bergson, 1991, p. 51)

Threads of Bergson's comments resound in Deleuze's attitude toward problems, by favoring the discovery and enumeration of problems over solutions by examining the consequences and many

المنارة للاستشارات
www.manaraa.com

becomings of the problem within its set of conditions. Problem-orientation can be seen in Deleuze and Guattari's (1977) turn toward 'machinic assemblages' and the Oedipalization of the subject through the triangle of 'mommy-daddy-me': on one hand, they sought to describe why a subject might want to be repressed within a socius, broadly outlining the problem of a subject's willing participation in a paranoid, if not outright fascistic system; on the other hand, the point of Anti-Oedipus is to confound clearly defined scopes of analyses, demonstrating (through a type of performance) how subjectivities are entangled within global and local networks of relations that are aligned and misaligned, collaborative and competitive, to demonstrate that problems emerging from life are messy. Consequently, Deleuze and Guattari were focused more on the processual ontologies of subjectivation, rather than a genealogical or historical epistemological approach. Desiring machines are repressed—but why? "To what end? Is it really necessary or desirable to submit to such repressing? And what means are to be used to accomplish this" (1977, p. 3)? The underlying problematic—the problem hinted at in the solution materialized within a state of affairs—is stated as, "[given] a certain effect, what machine is capable of producing it?" The point of a project like Anti-Oedipus, arguably, is that we can only see true problems if we entangle temporal and spatial analyses, allowing the results of our labor—expected or not—to come from the analysis of intersecting horizontal and vertical planes.

Deleuze and Guattari's work in *Anti-Oedipus* constantly questions the solutions a despot employs by iterating over the interaction of a problem with its conditions. When Deleuze urged us to stop treating problems as "givens" (1994, p. 159), he entreated us to discuss problems outside of their state of affairs, so that as many of the solutions produced by a problematic can be enumerated and explored. Specifically, Deleuze stated that "[even] if a problem is concealed by its solution, it subsists nonetheless in the Idea which relates it to its conditions and organizes the

المنارة للاستشارات
www.manaraa.com

genesis of the solutions" (1990, p. 54). Hence, like the problem of general computation, the solution is often mistaken for the problem, and misinterpreted through the by-product of a perspective that does not consider it problematically (Deleuze & Guattari, 1977, p. 3). Problems define the extent to which a solution can be, and they shape a state of affairs in myriad ways. Software solutions blossom within life's milieu. It is the task of a problematology to connect solutions to their actual problems, and to correctly interpret the effect its conditions have on its implementation.

De-conflating problems and solutions is difficult work, because conditions—like those Turing experienced in relating computer operations to the horizontal movements of tape reels—are intrusive. By following up their initial question linking effects to the machines producing those effects by asking, "given a certain machine, what can it be used for" (p. 3), Deleuze and Guattari (1977) offer a model for framing questions in terms of problematics. In doing so, they describe a process of problematization, where seemingly straightforward questions lead to the integration of many domains of knowledge. This is because problems are immanent to many types of conditions; problems cut across many scopes and boundaries, both social and machinic. What does it mean when the same problem has many solutions, or when the same machine is used to produce many effects? Does repetition in this way represent a failure to understand the problem, or complete dominance of it? This kind of questioning allows us to look beyond the actualizations of 'effects' within a state of affairs and begin working toward answering how and why those effects are evidence of a problem. And while the problem may not be fully understood, even by those entangled by the forces of production, like Facebook's software engineers asked to 'solve' the given problem of monetization, accurately connecting a solution to

المنارة للاستشارات
www.manaraa.com

it allows us to gain a better understanding of the true problem at the root of the efforts to solve them.

Problematics are confusing because of their experiential factors. Tape, at the time Turing devised his universal machine, made the most sense because it was materially feasible. Deleuze (1994) describes a part of experience as sense, which is "a condition of real experience, not possible experience" (pp. 153-154). Extracting a problem from its conditions is difficult because it requires one to destroy the "image of thought which presupposes itself and the genesis of the act of thinking in thought itself," because sometimes, "[something] in the world forces us to think. This something is an object not of recognition but of a fundamental encounter" (p. 139). Problems are inextricably linked to their conditions and mistaken for their solutions because conditions and solutions are typically experiential and actual, while problems are generally virtual. A problematic approach, through the successive encounter with the effect of a solution, yields some or all of the nature of the problem: the "ideational material or 'stratum'" defining its "condition of truth" yields itself as sense (Deleuze, 1990, p. 19), which itself "cannot be reduced either to the object designated or to the lived state of the speaker" (Deleuze, 1994, p. 154). To have a sense of a problem is to experience those conditions in a way that allows us to begin to explain the experience we have of that solution. But sense can be misleading: in the case of the Universal Turing machine, the problem of affecting work computationally is conflated with the machine itself, which is part of a solution that can be experienced directly. Problematic analysis allows this type of real experience to be teased *away* from the problems, in their virtual scope, by allowing things that can be experienced to be categorized as part of its conditions. This means that source code and programming are but part of the conditions of the problematic of software.

المنارة للاستشارات
www.manaraa.com

To close this section, Osborne's (2003) summary of Bergson's perspective on problematics is helpful because it describes a type of ontological investigation that incorporates the conditions a problem resides in into a processual form. Problems tend to exist in spite of their conditions. May (2005) described Deleuze's perspective on solutions as "a particular form of exhaustion" (p. 85), which is evident in Osborne's (2003) analysis of Bergson. Not only are problems the central aspect of life, according to Bergson, but that life exists in its many specializations is a demonstration of the problems it continually solves: "the contingency of problems and their local solutions" compound "in ever-multiplying webs of vital ordering and sub-ordering" (p. 6). Following this reasoning,

> an organ such as the eye is a solution to certain problems of action faced by the living being; but it is also something which is effective in the sense that it is capable of responding to future problems which are at present unknown. (p. 6)

The problems the eye evolved to address required open-ended solutions operating over grand scales of time; the eye itself is not a finalized structure and is therefore open to continuous evaluation. By focusing on Bergson's 'problems of the eye,' the emergence of a system that is anticipatory, rather than final comes into view. The eye, like software solutions, are contingent on their conditions and are open to re-evaluation. Does the eye continue to serve its purpose? Can it respond to new requirements, and conversely, what does it lack for? What happens when it is insufficient, or ineffective? Problematics describe how a problem is immanent to its conditions, and problems tend to be solved in multiple ways for multiple reasons. The important consideration Deleuze (and Bergson) recognized was that problems tend to be re-solved, iteratively, repetitively, with a degree of difference between those repetitions. This focus is explicitly compatible with modern software, which, until it is abandoned, at least, is an exemplar

المنارة للاستشارات
www.manaraa.com

of continuous delivery, of difference and repetition, of being maintained and re-released as bugs are found or features are added. Deleuze "regarded the analysis of problematics as being specifically a philosophical project" (Osborne, 2013, p. 4): problems may literally be the elephant's foot that scholars take turns touching, describing it in parts, as the analogy goes. A broad, expansive problem may assert itself within a locality and be misunderstood because its true extent and scope cannot be fathomed due to the many ways it is solved for and mediated by its conditions. Problematics is a way to follow actualized solutions back to their virtual problem.

This section explored Deleuze's (1995) definition of a problem, of a problematic, and the immanent nature of problems and the importance of their conditions. Not only was Deleuze and Guattari's (1977) concern for problematics evident in *Anti-Oedipus*, it went on to describe the role of the proposition in shaping the definition of 'science' in *What Is Philosophy?* (1994). Problematization is important for thinking about and through 'science' because it provides a means of orienting analyses of patterns and practices, like those of software engineering, not around a specific product, e.g., software, but through a broad array of activities and products. Problematization is at the root of discovering and exploring a scientific discipline's orientation to the world, its propositional encounters.

Problem-Orientation

This section builds on the last by emphasizing problem-orientation, which is a mode of inquiry proceeding from problematics that is evident in software engineering. This section describes how problem orientations emerge from a problem's material and experiential encounters with the world, which are propositional in nature (a concept defined in the next section). These encounters, in turn, describe how problems can orient agents toward solutions as

المنارة للاستشارات
www.manaraa.com

a negotiation of their conditions, and how problematics can give rise to the organization of a set of practices designed to overcome their conditions and solve their problems. This provides the basis for the conceptualization of a sociotechnical regime, and the role of problematics for those regimes. It begins by offering an example of how problems operate at different levels, demonstrating the importance of both Alan Turing's and Konrad Zuse's early concepts of general computation. Proceeding from a definition of 'problem,' it develops the idea of problem orientation by examining the ways in which Deleuze's theory relates to common software engineering practices. Problem orientation allows source code to be de-fetishized by recognizing that it is always generated in response to a given problem according to patterns and practices that exist because of the immanent, true problem. Problem orientation is a way of explaining how problematics organize the social and technical practices used to produce orders of knowledge—regimes--that generate solutions. Further, problem orientation allows for the delineation of true, 'virtual' problems, and 'given' problems, which arise from a state of affairs. This delineation provides a mechanism for describing how, for human agents at least, problematics begin organizing their solutions around regimes vested with social and technical practices, epistemologies, and ontological commitments.

Problems operate at multiple levels. Deleuze (1994) distinguished true problems from false ones, which differ in that false problems tend to be 'given' problems: while an engineer at Facebook was given the problem of capturing and recording a user's mouse movements efficiently, the 'true' problem might relate to the monetization of user attention. When digital computation became feasible, the problem of an 'instruction' was tied to an expression of a discrete numerical system, i.e., binary; what allowed 'computation' to move beyond specific tasks and into generality, capable of solving many things? Even if Alan Turing's tape described

المنارة للاستشارات
www.manaraa.com

to its machine how to perform the instruction, the machine still had to mechanistically perform properly ordered operations: because of the tape, it knew how to distinguish one operation from another, but the actualization of the machine had to solve another problem. In 1937, the German mechanical engineer Konrad Zuse recognized that, for a general computer, data and code are the same thing: if the conditions are a discrete set of electrical registers, the problem might be one of expressing and distinguishing 'data' and 'code' numerically (a topic further explored in Chapter 3). How does one operate on the other, and how does a universal computer distinguish them at the level of electrical impulses?

The nature of a problem affects the ways it is solved, which, on the face of it, is an obvious statement. However, the nature of problems has interesting ontological and epistemological considerations in Deleuze's philosophy because of his materialist orientation. For Deleuze, problems are immanent to their conditions, residing within them. Before diving into Deleuze's perspective, the lexical definition of 'problem' is worth revisiting: according to the Oxford Dictionaries, a problem is principally two things: "a matter or situation regarded as unwelcome or harmful and needing to be dealt with and overcome"; or "an inquiry starting from given conditions to investigate or demonstrate a fact, result, or law" (2017). Of the 32 synonyms Oxford Dictionaries lists, the common comportments are "difficulty, trouble, worry, complication"; the term itself, 'problem,' originated in late Middle English, and "originally [denoted] a riddle or a question for academic discussion." Presently, 'riddle,' in the academic sense, is nowhere to be seen in Oxford's list of synonyms for 'problem,' which is interesting in that problems are both calamity and equation in a Deleuzian sense. As May (2005) summarizes, Deleuzian "[problems are] an open field in which a variety of solutions may take place," where "[it] is the problems rather than the solutions that are primary" (p. 84). In this mode, solutions are

المنارة للاستشارات
www.manaraa.com

immanent to problems, and problems are immanent to a milieu; solutions represent a type of difference and repetition—or iteration—on a problem. “[Solutions] are a particular form of exhaustion,” while problems “are inexhaustible” (May, 2005, p. 85), implying that, for Deleuze, solutions are actualizations, while problems are virtual, in the sense that they are fields of potential. Ontologically, this implies that that solutions are expressed as localizations, individuated by the differences of their repeated encounters with a given problem. So, in the ‘given,’ or false sense of the problem Deleuze described, there is an agent that must produce a solution that satisfies a set of requirements; in the ‘true’ sense, the problem is immanent to the milieu in which it is embedded, reflexive of that milieu, and open to new solutions. In this way, solutions become a way of knowing a problem (or an aspect of a problem) by providing a sense of it.

The confounding strength of Deleuzian problems is their immanence and virtualness, because it is easy to conceptualize them in terms of transcendence, which is inappropriate. Problems are fundamentally comprised of visible and invisible elements, i.e., actual factors that can be enumerated and therefore anticipated, and virtual factors which remain to be discovered. Deleuze (1994) stated that “[problems] are tests and selections” (p. 162), which is of fundamental import, because it allows solutions to become a type of probing for a problematic. Osborne’s (2003) description of Bergson’s problem of the eye described it as a solution for “certain problems of action” that was also “capable of responding to future problems” which cannot be known until they are encountered (p. 6); as Deleuze’s perspective on problems was influenced by Bergson, what distinguishes it most from traditional dialectical problematics is his emphasis on processual, continuous encounters between a problem and a locality (Deleuze, 1990; 1994; May, 2005; Koopman, 2013). Deleuze used swimming to illustrate a problem’s

المنارة للاستشارات
www.manaraa.com

encounter with an event's conditions: "[to] learn to swim is to conjugate the distinctive points of our bodies with the singular points of the objective Idea in order to form a problematic field" (1994, p. 165). Learning to swim is a means of interrogating the "problem projected" by the encounter between it and a body; it is one actualized solution to "the complex theme which does not allow itself to be reduced to any propositional thesis" (Deleuze, 1990, p. 122). A problem is an expression of the virtual by being fields of potential, and changes according to the conditions from which it is immanent; when a problem is identified, it is determined by "means of an appropriate process … in space and time" which, "as it is determined, … determines the solutions in which it persists" (p. 121). Ultimately, a solution is the "synthesis of the problem with its conditions" which "engenders propositions, their dimensions, and their correlates" (p. 121). A solution is one actualization of a problem at a specific point in space and time, and problematics describe a way of thinking about how the repeated encounters between a problem and its conditions lead to new solutions.

Chaos (or the virtual, the space of possibilities or the problematic field) ensures that no problem is solved with finality, which is the major contribution Deleuze and Guattari (1994) made in their final collaboration. By interpreting the working differences of science and philosophy as fundamentally one of their orientation toward chaos, e.g., the planes of reference and immanence, of sieves stretched across the virtual operating at different speeds, Deleuze and Guattari folded the nomadic into any actualized solution. It is a difference of 'solving' rather than 'solved': in Software Engineering, for instance, a problem must often be 're-solved' due to a phenomenon called 'scope creep.' Scope is a term used to describe the amount of work that must be done to meet the design requirements of an artefact; creep is a verb used to describe an effect on the work performed in accordance with a design. At some point, a design is "finalized,"

المنارة للاستشارات
www.manaraa.com

and work begins. Scope creep is a pejorative term for a change in design that happens after work has begun: if the design changes, the problems of today may be entirely different from the problems of yesterday, which can lead to work being thrown out. However, scope creep is a natural product of a problem being synthesized with "its conditions" (Deleuze, 1990, p. 121). A change in design can be, like in Brooks (1995) case of the many starts and stops of IBM's OS/360 platform, the result of discarding 'false' problems after a series of "tests and selections" (Deleuze, 1994, p. 162) from the 'true':

> The only way to take talk of 'true and false problems' seriously is in terms of a production of the true and the false by means of problems, and in proportion to their sense. … Not only is sense ideal, but problems are Ideas themselves. (Deleuze, 1994, p. 162)

Code produced for a software engineering effort is constantly 're-factored,' reconciled, and rewritten in response to a change in the scope that determines the extent to which software 'solves' its given problem. As Kohanski (2000) states, "[a] designer will try to anticipate … midstream changes and build in the flexibility to handle them, but no one can anticipate everything," explaining that some "code will be bent to fit the new requirements and some will be left alone" (p. 176). Work toward a software engineering solution is a processual thing, a constant refitting of connections between old understandings and new requirements. Software ontologies emphasize dynamism, in the sense that new versions replace old versions in major and minor ways: a product may be rewritten from scratch, have new features patched into it, or receive maintenance releases to fix bugs. At some point, however, a solution must be measured in terms of its problem: "[did] we really want it to do the things we designed it to do" (p. 175)? How do we know that what we designed has any relation to the problem instantiating the process

المنارة للاستشارات
www.manaraa.com

of its implementation? Rather than delineate by 'true' and 'false' problems, Deleuze distinguished them by 'virtual' and 'given,' a distinction that provides a means to identify and relate one or more solutions to the nature of their problem. A given problem is specific to its set of conditions, which may relate to a specific feature in an application; the underlying problem that the broader software application attempts to 'solve' may relate to a true problem, which because of its virtuality, invites many solutions by always being open to new actualizations.

By refusing to treat problems as true or false, Deleuze (1994, 1990) argued that 'true' or 'false' were a matter of a solution's production: there are true or false solutions. Rather, the significance of 'virtual' and 'given' then is one of perspective: a solution becomes measured by the degree to "which propositions correspond" to "particular responses" within a locality (1990, pp. 120-122). Solutions are a way of knowing a problem, and problems are both a way of knowing an impetus for being, for looking at solutions as expressions of a virtual or given scope of conditions. As stated earlier, "solutions are engendered at precisely the same time that the problem determines itself" (p. 121); how a solution is designed, as Kohanski (2000) lamented, allows one to parse its manifestation (e.g., software) from the problem (or problems) it is designed to solve. This delineation provides a means for locating practices of reading source code or analyzing software itself on the side of 'given' problems, while examining the processes of its design and implementation as attempts to actualize an aspect of a 'virtual' problem.
It is the difference between attempting to read meaning out of a variable name, which as a 'given' issue may have little to no bearing on the virtual purpose of the software itself, and the ways in which software systems mobilize psychological considerations to subjectivate users for purposes of monetization, which is a 'virtual' problem that continues to find new actualizations because of capitalism. In this way, the many types and layers of 'given' problems, from a

www.manaraa.com

variable name, a class method for tracking mouse movements, to a computer language's compiler design, and a computer architecture's affordances, can be seen as 'givens' in the scope of a virtual problem's continual engagement with its conditions. The problems of computation are virtual, while those of implementation are givens. Alan Turing's argument for a Universal Machine straddled a line between the inexhaustibility of a problem in its virtuality and the necessity of imagining its solution in terms of the conditions given to him. The point of contact between 'virtual' problems and their actualizations are systems of knowledges and processes that are designed to interpret and implement them, called sociotechnical regimes.

This section has described how problem orientations not only emerge from a problem's material and experiential encounters with the world but produce the sociotechnical regimes by which those problems are solved, be it the problem of the 'eye' to see what is necessary to be seen, or a Universal Machine to account for the necessity to generalize a computational solution across any set of numbers. Problematics and problem-orientation describe how a discipline can be organized around the principles and practices focused on solving them. The next section outlines and explores the Deleuze and Guattarian (1995, 1987) basis for considering a discipline as a distinct set of practices which navigate the 'virtual' and 'given' qualities of a proposition to produce their solutions.

Sociotechnical Regimes

This section explains what sociotechnical regimes are by connecting the previous sections' discussion of problematics to Deleuze and Guattarian (1995, 1987) concepts. Sociotechnical regimes describe how and why propositions are answered in the ways that they are, which allows a product like software to be seen as less for its programming or source code,

www.manaraa.com

and more for its processual design, development, and delivery. Software engineering's 'virtual' problem describes circumstances in which 'problems are solved computationally' are a smooth space because it is open and continuous and chaotic; it is only through its solving, e.g., by producing software, that its solutions can be seen as a process by which many 'given' problems are solved at many material levels, through many social and machinic negotiations. It is important to understand software as a product of a regime and not as 'programming' or 'source code' alone. Software engineers are more than just programmers, and programmers do more than just program. Engineers of any kind represent an intersection between social and machinic relations, and work to translate problems from social domains into technical ones, and vice versa. Software is a peculiar response to propositions that ask to solve problems computationally; software engineering, as a sociotechnical regime, describes the efforts and organizing principles that go into a proposition's response. The theories of Deleuze and Guattari, singularly and conjunctively, are suited for a description of software engineering because of the flexibility of computation's underlying material nature: software engineering is a harness on a peculiar type of chaos, which is a convenient way of explaining why software has historically been unreliable and unpredictable.

What is a Sociotechnical Regime?

Deleuze and Guattari recognized the interplay of axiomatics, structures, and modes to combat immanence, disorder, and chaos: "[what] had interested Deleuze from the beginning was the ways in which order comes to be maintained in the world despite [an] immanent threat always knocking at the door" (Arnott & DePaul University, 1999, p. 50). Deleuze and Guattari were accounting for science's entanglement with social and cultural factors in their discussion of

المنارة للاستشارات
www.manaraa.com

propositions and functions—regimes live in a constant state of disruption. The plane of reference only has the appearance of stability; it is within the self-interest of a technical regime to produce consistency. The concept of the sociotechnical regime is a way to describe consistency in the face of chaos. Sociotechnical regimes are a way to describe the processual relations that lead to the production of a scientific plane of reference, which allows the regime—as an organized entity and community of practice—to explain and solve their motivating problematics. Regimes relate specific social and machinic relations to propositions and functions, which both identify them (e.g., 'Computer Science,' 'mechanical engineering,' 'civil engineering') and cause them to relate to broader, externally motivating factors in specific ways. Regimes operate and perpetuate themselves by being perpetually and processually engaged assemblages of human and technical agents that maintain their internal order by modulating their structures and modes in response to internal disruptions and external motivations. Order in a sociotechnical regime is enforced so that subjects are produced according to the axiomatics of their episteme; order flows from the axiomatics used to employ the functions and propositions encoded on their disciplinary plane of reference. This section offers a top-down explanation of the sociotechnical regime concept, defining a plane of reference, a function, and a proposition.

Sociotechnical regimes are identities oriented around a volatile area of a scientific plane of reference. In *What is Philosophy?*, Deleuze and Guattari (1994) work to delineate philosophy, and what philosophy does, from what the arts and sciences do. Each "thought-form" (Arnott, 1999, p. 49)—science, philosophy, and art, a topic not taken up by this dissertation—organizes what it knows and uses on planes: science uses a plane of reference; philosophy uses a plane of immanence. The task of Deleuze and Guattari's (1994) last collaboration is relevant to sociotechnical regimes, not only by pulling apart the disciplines along lines of production to

www.manaraa.com

show how each discipline performs its work, but by arguing that the knowledge and ontologies embedded in a plane of reference or immanence are used to make sense of chaos. A plane of reference is defined as "a fantastic *slowing down*, and it is by slowing down that matter, as well as the scientific thought able to penetrate it with propositions, is actualized" (p. 118). Functions are "a Slow-motion," which direct affect matter. By slowing things down, a plane of reference imposes "limits or borders" through which science "confronts chaos" (p. 119). Propositions are a form of "reference" which is a "relation to a state of affairs to the system," which serves to relate functions to a problem or task (p. 122), which is better explained as, "[the] object of science is not concepts but rather functions that are presented as propositions in discursive systems" (p.117). Unpacked, this means that propositions relate functions to a question or task. Philosophy relies on infinite speed so that thought can traverse and find relations between concepts across a plane of consistency: concepts must be free to relate to each other. Science, on the other hand, first relies on a "set of coordinates" to which a "relationship" of a variable forms an "endoreference": there is quantification at the core of a plane of reference that refers back to an immanent reality. Despite the work and products of philosophy and science being different, Arnott's (1999) summary of Deleuze and Guattari's work concluded that "any attempt to blur the boundaries" between the different thought-forms of science, philosophy, and art "is to be resisted," so that instead scholars might look at the ways they interfere with each other (p. 49). Philosophy creates and works with concepts; the arts create figures using prefects and affects; and the sciences generate prospects using propositions and functions (p. 24). Science does not create concepts; philosophy does not create functions and propositions. But, they can meet somewhere in the middle, when the products of their efforts contact each other in the world. Put another way:

المنارة للاستشارات
www.manaraa.com

> philosophy explores the plane of immanence composed of constellations of constitutive forces that can be abstracted from bodies and states of affairs. It thus maps the range of connections a thing is capable of, its “becomings” or “affects.” Science, on the other hand, explores the concretization of these forces into bodies and states of affairs, tracking the behavior of things in relation to already constituted things in a certain delimited region of space and time (the “plane of reference”). (Smith & Protevi, 2015)

The adjoining concepts between philosophy and science of bodies, ‘becomings,’ and states of affairs, are crucial to developing an understanding of how problematization orients sociotechnical regimes, and why philosophy—the discovery of connections between things—can be a method for describing how regimes come to be and perpetuate themselves. Isabelle Stengers (2005) focused on how concepts and functions might intersect in *What is Philosophy?*, concluding that they do so “only after each has achieved its own specific self-fulfillment” (p. 151), noting that Deleuze and Guattari (1994) stated that intersections occur “in their full maturity, and not in the process of their constitution” (p. 161). Such thinking implies that concepts cannot develop a function, and vice versa; concepts have nothing to add to the creation of a function, because concepts do not belong in the domain of the function. Similarly, science attempting to create concepts by describing them with functions is inappropriate. The problem-oriented sociotechnical regime is a conceptual framework that encapsulates the products of a scientific discipline’s efforts to produce functions to solve problems, while recognizing that those problems constitute the basis of the regime’s existence.

Separating concepts from functions traces endoreferences, external points of contact between the things a regime creates and the principles and practices behind their creation, which for the sciences are propositions. Sociotechnical regimes describe and explain how, in the case of

www.manaraa.com

software engineering, software is produced, how software engineering is different than Computer Science, and why programming and source code are one function of many. By the time the first chapter of *What is Philosophy?* concludes, the authors identify an enfoldment of philosophy into science, and unfurl it over the course of their project. This enfoldment is evidenced by their assertions that while concepts are for philosophers alone to create, they are incorrectly credited to the sciences, e.g., "[the] power of the concept is attributed to science, the concept being defined by the creative methods of science and measured against science" (p. 33). But as Deleuze and Guattari argued, the opposite is needed: it is necessary to show why an enfoldment of one into the other is wrong. A philosophy focused on concepts, of their connections and becomings, is itself a method to understand science and its discursivity by demonstrating the operations of functions based on relations of social and machinic relations. Sociotechnical regimes define an area of a plane of reference, of how a peculiar set of social and machinic relations works to solve an underlying problematic. The concept can parse the liminal areas between disciplines, like Computer Science and software engineering, accounting for both the differences and remaining connections, while exposing areas the functions and propositions of a discipline can become something else in relation to the concepts of media studies and other philosophical disciplines.

Propositions are important to understand because they are, in many ways, an almost dialogic process by which a sociotechnical regime produces itself. Sociotechnical regimes initially organize themselves around problematic, and then produce themselves if their solutions are valued by outsiders: think, for instance, of the technical identities that no longer exist. Propositions are interpreted by a sociotechnical regime in specific ways, ways that are not unlike what Foucault described as a "regime of truth or error" (2003, p. 164). Essentially, regimes are a

www.manaraa.com

type of “grid of intelligibility” that defines the “division between truth and error.” A sociotechnical regime determines the principles for how the ‘correctness’ of an algorithm can be measured in conjunction with a problem and determines how that solution can be valid or invalid within the scope of its actualization. What delineates a technical regime from the broader definitions of ‘discursive system’ or ‘regime of truth or error’ is outlined in the specificity of an assertion Foucault made while explaining how processes of actualization came to be discernable as systems: at some point, “[a] corpus of knowledge, techniques, [and] ‘scientific’ discourses” were “formed and [became] entangled with the practice of the power to punish” (1995, p. 23). A sociotechnical regime internally regulates its truth-values and its methods of growing to encapsulate new functions, while being used by external forces to move the present into the actual: while regimes are governed by their orienting problematics, they come into being and persist because the solutions to that problematic has value.

Sociotechnical regimes have certain ways of interpreting and acting on the problematics that lead to their creation, in the ways they relate propositions to functions, problems to solutions. They act like identities for their agents and delimit the ways those agents relate to outsiders. Where Deleuze and Guattari use ‘discursive system’ to broadly describe systems of actualization, it is helpful to revisit Foucault’s (1982) concept of the ‘episteme,’ because it is helpful for reinforcing the idea of a sociotechnical regime as an organized set of social and machinic relations, oriented around a problematic. He wrote that an

> episteme is not a form of knowledge (connaissance) or type of rationality which, crossing the boundaries of the most varied sciences, manifests the sovereign unity of a subject, a spirit, or a period; it is the totality of relations that can be discovered, for a given period,

المنارة للاستشارات
www.manaraa.com

> between the sciences when one analyses them at the level of discursive regularities. (p. 191)

An episteme is not knowledge itself but is rather that through which something emerges, and Deleuze and Guattari's (1994) argument that the thought-forms are principally ways to account for chaos are crucial for understanding how sociotechnical regimes flow from problem-orientation. While regimes are in many ways self-organizing beyond the initial conditions that brought the confluence of factors that would comprise them, they are only perpetuated if they can successfully modulate themselves in response to immanent disruptions, i.e., adapting to chaotic and changing circumstances. The obvious way this adaptation is performed is through the production of actualizations which are valued by external motivators. Foucault's concept of 'episteme' becomes increasingly valuable for thinking through the implications of Deleuze and Guattari's thought-forms: why, for example, did medical sciences reorient themselves in the span of 20-years, virtually overturning (or reorganizing) their prior plane of reference? A response to this question can be found in discussing assertions made by Levy-Leblond which Guattari (1984) uses to frame relationships of subject positions (i.e., "agents of change") that are modified by degrees and intensities of translation of their position within a totality (or assemblage), demonstrating the principles informing the usefulness of mathematics and its practitioners. Mathematics "may have a relationship of application," such that it is used by disciplines like biology or chemistry to quantify values; it "may have a relationship of *constitution* or *production*" (emphasis maintained, p. 122). Application and production are actions which describe how sociotechnical regimes operate, in that the corpus of their techniques and knowledge are practiced (applied), so that something specifically valuable is actualized (produced). Agents of a sociotechnical regime can be thought of, as Foucault (1995) argued, an

www.manaraa.com

"element in which … the effects of a certain type of power and the reference of a certain type of knowledge" are produced (p. 29). In turn, they produce and further "the machinery by which the power relations [of their formation] give rise to a possible corpus of knowledge, and knowledge extends and reinforces the effects of this power." Essentially, in response to being 'useful,' a technical regime produces its internal agents of change, while being, in and of itself, an agent of change in constant interaction with external agents.

Functions are a sociotechnical regime's ability to operate on the real, in the sense that the real is materiality. Sociotechnical regimes solve problems according to the limits and affordances of the functions and propositions encoded on their area of their scientific plane of reference. At the root of any problematic lies an observer, who not only notices its existence (c.f. Koopman, 2007), but has a peculiar set of discursive systems designed to "solve" it, to probe it, to enumerate its components and implications, to begin to realize its contingencies, and in the case of software engineering, to begin the task of actualizing solutions for the problem by producing concrete responses to the virtual problem of computation. The owner's discursive system determines what is discoverable and how it will be dealt with. Deleuze and Guattari (1994) cite Foucault's definition of the 'actual' in explaining how things come to be, how they become. Bodies are somehow produced as arrangements (concrete assemblages) that are the "difference between the present and the actual" (p. 112); bodies are "in the process of becoming," and are actualizing in some way. The difference between the actual and the present, then, is that by becoming actual, a body ceases to be 'in the present,' it is "what already we are ceasing to be." In this way, Deleuze and Guattari argue that Foucault believes "the object of philosophy is not to contemplate the eternal or to reflect on history but to diagnose our actual becomings." They further describe a system in which concepts are valued for their own creation and for the

المنارة للاستشارات
www.manaraa.com

discovery of their pure events, explaining how concepts form connections to other conceptual milieus (which are diagrammatic) on the plane of immanence, and why this recognition matters (p. 31; p. 36). Concepts are brought into being according to a philosophical episteme; propositions and functions, conversely, are brought into being according to a scientific episteme. Philosophic problems are under the same constraints as scientific problems, in that, be it by plane of immanence or plane of reference, the means of identifying a problem within its conditions are shaped and delimited by their limits of their respective plane. Perspective, the ability to see the conditions in which a problem emerges, is organized by a discipline's plane.

While functions are created, revised, and discarded by a sociotechnical regime when they gain new insights into the material nature of a problem, regimes need new propositions to create, revise, and discard functions. Because of their orienting problems, sociotechnical regimes have relatively stable identities and idiosyncratic ways of problem solving, but they do change over time. The last point that Foucault made which is relevant for epistemes is that "the rhythm of transformation doesn't follow the smooth, continual schemas of development which are normally accepted" (Foucault, "Truth and Power" in Foucault & Gordon, 1980, p. 113), which is important to keep in mind, because science does not always follow a linear progression. Regimes adapt and change as the conditions of their problem are better understood, expanded to incorporate new insights, or limited for the same reasons. This is where the concepts of Deleuze and Guattari (WIP, TP, AO) are particularly suited to producing a type of history that allows observers to keep track of the immanent social and machinic relations encapsulated by a problem-oriented sociotechnical regime: a plane of reference comprised of functions designed to respond to propositions describe a plausible way the sciences relate to the world.

المنارة للاستشارات

www.manaraa.com

How does a Sociotechnical Regime work?

What distinguishes sociotechnical regimes from other ideas used to describe disciplines, like communities of practice, discourse community, or Kuhnian paradigms, is the inherently open-ended nature of Deleuze and Guattari's thinking, which focused on the always 'becoming' nature of the relations and movements of immanent relations. As Kuhn (1996) might explain, the history underlying the discipline of Computer Science (CS), as the Association for Computing Machinery (ACM) would desire it, has "been drawn, even by [the] scientists themselves, mainly from the study of finished scientific achievements" (Kuhn, 1996, p. 1). Reliance on such a history, despite its intent to be "persuasive and pedagogic," fails to adequately represent a discipline's attempt to capture "the enterprise that produced" those achievements. Consensus and success are emphasized, rather than failure or controversy. By discarding those adjacent factors, the human and material trends that detract from the sweeping (and linear) narrative tend to be overlooked or ignored, such that traditional historical methods not only have "difficulties in isolating individual inventions and discoveries," but give reason "for profound doubts about the cumulative process through which … individual contributions to science were thought to have been compounded" (p. 3). A Deleuze and Guattarian perspective seems tailor-made for capturing and accounting for the exclusions a discipline makes as it organizes its "commitment[s]" around an "apparent consensus" about its "normal science," which are predicated on "one or more scientific achievements" (Kuhn, 1996, pp. 10-11). The concept of the sociotechnical regime draws together Deleuze and Guattari's (1994, 1987) concepts of the event, plateau, rhizome, smooth and striated spaces, and aforementioned plane of reference, propositions, functions, and problematics into a framework that undermines paradigmatic thinking by incorporating failure into its narrative, and difference rather than enfoldment, interference rather than autonomy.

المنارة للاستشارات

www.manaraa.com

Sociotechnical regimes offer a historical perspective on the development of a discipline, field, or area of technical practice. A crucial factor in Deleuze and Guattari's (1994) thinking was their emphasis on historical effort, for there are no problems of science or of philosophy that are not historical, that are not driven in some way by contextual elements (c.f. "geophilosophy" and Nietzsche's work to describe national characteristics, pp. 102-103), or by intrusions of a state of affairs upon an agent of change. Virtual problems are not 'given to us'; rather, we work to discover the problem that is obfuscated and submerged by a state of affairs—it is in the problem's transcendence of a milieu that problematology becomes an interdisciplinary concern. Thus, the work outlined in *What is Philosophy?* is significant: in some cases, the problems of science have answers in philosophy, and vice versa; problems are inflection points and intensive events representing points of opportunistic intersection between concepts and functions. The concept of 'intelligence,' for instance, is being pressed upon and availed by machine learning and artificial intelligence, but its problems traverse and 'survey' the planes of immanence and reference in much the same form. Problematics are the bridge that will demonstrate the differences and commonalities between disciplines while accounting for their organizing principles, which is how and why they do what they do. Problematics, for media theories of software, allow the processes and practices involved in the work of producing that software to be accounted for.

Events

At the root of any sociotechnical regime is a problematic encounter that lead to its organization, called events. Events represent a point in time where something happened and are the initial point for mapping where one regime ends, and another potentially begins. "Events are

المنارة للاستشارات
www.manaraa.com

ideal," Deleuze (1990) explained: they are "ideational singularities which communicate in one and the same Event. They have therefore an eternal truth, and their time is never the present which realizes them and makes them exist" (p. 53). Meaning, they tend to happen and then be noticed as having happened. Two water-based examples from Deleuze's (and Guattari's) work serve to illustrate how a problem emerges through an agent's encounter with an event: "[it] was at sea that smooth space was first subjugated and a model found for the laying-out and imposition of striated space, a model later put to use elsewhere" (emphasis added, Deleuze & Guattari, 1987, p. 480); and, "[to] learn to swim is to conjugate the distinctive points of our bodies with the singular points of the objective Idea in order to form a problematic field" (emphasis added, Deleuze, 1994, p. 165). Through an encounter between bodies, an understanding emerged that a problem existed. The problem, in the first example—the smooth space of oceanic water—emerged that drowning was a real consequence of being submerged by that encounter. Through repeated interactions, a model of buoyancy emerged, leading to the development of 'new bodies,' like boats and triremes, to striate the smooth space. The second example Deleuze (1994) offered demonstrates how problems are always immanent to encounters between bodies. The water-based examples provide a basis for understanding problematics as processual, in that the problem is repeatedly encountered and reflected upon while its conditions are explored and accounted for.

The creation and maintenance of a sociotechnical regime can result from one or more events. Events are a point in time that mark a mode of thinking about a problematic. For example, one would be hard pressed to relate the event signified by the Sumerian abacus to the discipline of Computer Science, or the creation of numbers, which are fundamental to so many areas of knowledge. Martin Cambell-Kelly et. al's (2014) popular history of the computer

المنارة للاستشارات
www.manaraa.com

focuses not on the creation of digits, or of writing and inscription technologies, but on how humans performed the work of creating logarithmic and trigonometric tables from the sixteenth-century onward (pp. 3-5). Cambell-Kelly et. al demarcated some events from others by focusing on the relations of a sufficiently advanced form of mathematics—like actuary tables—to aid in the configuration of bodies for the purposes of seafaring. The problem of computation came about, in their view, when the need to "organize information processing on a large scale" became pressing (p. 3). At its onset, the problem of computation intersected with the practical need to aid ship-based navigation, which necessitated a set of social and machinic practices around computation, i.e., figuring numbers and probabilities, creating actuary tables, inscribing, storing, and recalling values, validating results, etc. An impetus in the sixteenth-century—an endoreference—began organizing a regime around a problem and its conditions.

Events are revisited repeatedly, and a sociotechnical regime constantly revisits those events, even if it attempts to forget them (e.g., Kuhnian's normal science). Deleuze and Guattari (1994) stated that "new concepts must relate to our problems, to our history, and above all, to our becomings" (p. 27), asking "[what] is the philosophical form of [a problem] of a particular time?" (pp. 27-28). Events represent a point in time where propositions ask for functions that can provide responses. If the functions to respond to an interposed and intrusive problem do not exist, either historically or within a state of affairs, they are created, if the conditions for the problem allow their immanent creation (p. 133). Cambell-Kelly et. al's (2014) work moves quickly over centuries of human-computational labor practices, toward Charles Babbage's efforts to finish the work begun by Baron Gaspard de Prony, a Frenchman who was tasked with automating the creation of actuary tables used for taxation and economic purposes during the French Revolution. Babbage acquired funding from the British government in 1823 to build the

المنارة للاستشارات
www.manaraa.com

Difference Engine and faced the daunting task of having to both design it, and "[develop] the technology to build it. Although the Difference Engine was conceptually simple, its design was mechanically complex" (p. 7). Although the Difference Engine was never fully realized, by struggling through the conditions of the problem of 'automation' for removing human labor from the task of computation, Babbage gained insights into a machine he called the Analytical Engine, which "would be capable of performing *any* calculation that a human could specify for it" (p. 8). This machine predates the paper Alan Turing published outlining the Universal Turing Machine by 102 years. Events are evidence that problems, for Deleuze and Guattari, operate at many levels and shape the conditions through which they are re-encountered.

Plateaus and Rhizomes

Plateaus represent a span of time progressing from one or more related events in which encounters with those events continue to affect the becoming of things. A sociotechnical regime can be thought of as a period of time leading from one or more related events encompassing "flows of varying speed and slowness" for humans that occur "alongside other planes of becoming," such as "animals, machines, molecules and languages" (Colbrook, 2002, p. 66). Deleuze and Guattari (1987) explain that plateaus are "always in the middle, not at the beginning or the end," and that a "rhizome is made of plateaus" (p. 21); further, "a 'plateau'" is "any multiplicity connected to other multiplicities by superficial underground stems in such a way as to form or extend a rhizome (p. 22). Treated as a span of time, plateaus capture and group encounters leading to and from one or more related events. Plateaus, for sociotechnical regimes, represent a period of time around encounters with a problem; the use of the concept allows analyses of regimes to incorporate formative and tangential encounters to discover ways in

المنارة للاستشارات
www.manaraa.com

which a problem is revisited over time, despite changing conditions. As many plateaus can exist simultaneously and overlap—due to their rhizomatic properties, which will be explained shortly—plateaus are a mechanism that allows software engineering to be delineated from Computer Science. They encounter the same problems but diverge in the ways they interpret them and produce actualizations from those encounters; the problems of one regime can be the problems of another, but their values and methodologies can diverge for reasons discoverable in their state of affairs.

If plateaus describe a period of time around an encounter with an event, rhizomes describe all of the social and machinic connections made between agents operating in that time span, around those events. Rhizomes, according to Deleuze and Guattari (1987), are not a root system in the traditional sense: they allow "any point" to connect to "any other point" (p. 21). These connections are significant for understanding sociotechnical regimes because they "bring into play very different regimes of signs, and even nonsign states" by capturing both signifying and asignifying connections, a process described in Chapter 4 of this dissertation. They work within a plateau because they are "composed not of units but of dimensions, or rather directions in motion," having "neither beginning nor end, but always a middle (*milieu*) from which it grows and which it overspills." Rhizomes both "[operate] by variation, expansion, conquest, capture, offshoots," and have no hierarchy, no traditional taxonomy. They are significant because, from a historical perspective, they implicitly resist linearization—Kuhnian "normal science"—by decentering authority and traditional explanations. Rhizomes help a sociotechnical regime capture the states from which it emerged and the relations that give it a shape in respect to its problematics.

المنارة للاستشارات
www.manaraa.com

A point Deleuze and Guattari (1987) make about rhizomes would seem to contradict the 'organizing' nature of sociotechnical regimes and their later focus on planes of reference: they state that rhizomes do not have an "organizing memory or central automaton," and are rather a "circulation of states" (p. 21). The purpose of a rhizome is capture movement over time, hence the focus of plateaus as being in the middle, of rhizomes as expressions of a milieu. The point of the rhizome is that it is not a map in a cartographic sense, but a projection of a state that "is always detachable, connectible, reversible, modifiable, and has multiple entryways and exist and its own lines of flight." Rhizomes do not contradict the concept of the plane of reference, because the openness of a rhizome, its lines of flight, capture meaningful innovation, where functions created and are connected to propositions to produce new actualizations that could not exist before. For example, the solutions afforded by the discovery of the "planar process" by Jean Hoerni in 1957 lead to a method of fabricating silicon transistors that is still used today. The planar method Hoerni developed increased the reliability of transistor-based designs, allowed transistors to be printed onto silicon wafers, and "made it easy to interconnect neighboring transistors on a wafer," which had the effect of "rendering the competition's offerings obsolete" (Riordan, 2007). Jean Hoerni was hired out of academia in 1956 by William Shockley, of the Shockley Semiconductor Laboratory to "do theoretical calculations of diffusion rates"; Shockley was shunned by other researchers at the time for "his pursuit of difficult R&D projects at the expense of useful, salable products." While Hoerni was initially isolated from other employees, he "kept coming around and snooping in the lab in the main building," which gave him "valuable insights into solid-state diffusion." His connections and relations in Shockley's lab would later give him the ability to follow a line of flight: "In a loose, fluid scrawl interspersed with three simple drawings," he created "a revolutionary new way to fabricate transistors—unlike anything

المنارة للاستشارات
www.manaraa.com

ever before attempted." In response to a proposition encapsulating issues of simplicity, reliability, predictability, and reproducibility—things vacuum tubes could not offer, and contemporary transistor production failed at—Hoerni's innovation was the product of social and machinic relations that produced new functions for a proposition leading to "a manufacturing method rooted in the mechanical printing process originated by Johannes Gutenberg more than 500 years ago." A function on a plane of reference can connect to any proposition, despite of its distance from the contemporary milieu, and not only that, a proposition can use any function on any plane of reference. In this way, the benefit of thinking about disciplines in terms of sociotechnical regimes, as plateaus and rhizomes, is that it opens and blurs disciplinary boundaries by accounting for lateral movements and social pressures derived from their milieus, their state of affairs.

Smooth and Striated Spaces

Thinking about sociotechnical regimes in terms of plateaus and rhizomes firmly roots them in a milieu, as products of a problem's conditions, as responses to many propositions, and as systems that produce idiosyncratic responses to those propositions. But to appreciate the products of a sociotechnical regime—like the aforementioned silicon transistors printed using planar methods in the spirit of a Gutenberg press, or the software 'software engineering' produces—it is necessary to understand how a regime intersects with its milieu, which explains how it makes sense of it, and how it interprets the propositions it encounters. Smooth and striated spaces are Deleuze and Guattarian (1987) concepts that describe how meaning is shaped by one's perspective. "Striated space," they explain, "is canopied by the sky as measure and by the measurable visual qualities derived from it" (p. 479). Meaning in striated space is construed by

المنارة للاستشارات
www.manaraa.com

reference, derived from a point's position relative to other points from the perspective of an observer. Striated space is organized space. An example of striated space is evident in how computers represent numbers using binary, or base 2: one bit represents 2 values, 0 and 1; the combination of two bits represents a set of 4 values, 0, 1, 2, and 3 (00, 01, 10, 11); 8 bits represent 256 values, from 0 to 255 (starting at 00000000, ending at 11111111). The meaning of the combination of values is relative to the system that encode and decodes those values and is thus *derived* from that relation.

To describe smooth spaces, Deleuze and Guattari (1987) use sound, because it is continuous, so when humans encounter it, they interpret it through their body, feeling it to understanding it. They state that "[the] smooth is a nomos, whereas the striated always has a logos, the octave, for example," implying that music is both law and spirit (p. 478). Smooth space is "filled by events or haecceities," and "is a space of affects, more than one of properties," and is more often felt than seen. Smooth spaces are continuous and unorganized because, from the perspective of an observer, they cannot be separated into distinct elements or forces while they are being felt: they are not governed by points and measurements. An important example of smooth space Deleuze and Guattari use is the ocean, where currents flow in directions governed by vectors, rather than by specific points in space (pp. 478-482). The concepts of smooth and striated spaces recognize that meaning is relative to a perspective, and the significance of smooth and striated space for a sociotechnical regime is that meaning, at the intersection of a proposition, becomes a negotiation. Some propositions are not immediately compatible with a regime's ways of becoming. When they are asked to consider a proposition that is more nomos than logos, they must work according to their methodologies to make sense of the proposition, and to effectively turn what could be the spirit of a proposition into the logos the regime employs

المنارة للاستشارات
www.manaraa.com

to discover and apply the proper functions for the expression of the proposition's solution. The negotiation of smooth spaces at the level of the proposition connects sociotechnical regimes to Guattari's (2011, 1984) mixed semiotics model, and is a topic explored in Chapter 4. What matters here is that Deleuze and Guattari's (1987) concepts of space describe how sociotechnical regimes act relative to themselves and others. The point here is that smooth and striated spaces conceptually allow source code and programming to be considered as functions within a broader mechanism that values processes over product by recognizing that a broader logos connects the many aspects of a rhizomatic set of social and machinic relations together on a plateau that continues to be in the middle, in the milieu of society and culture.

Sociotechnical regimes are a way to relate problematics to a discipline and its practices using Deleuze and Guattarian concepts. A problem is always, from Deleuze's (1995) perspective, as an immanent product of its conditions, and problems change according to their milieus. For the purposes of this dissertation project, Deleuze and Guattari's (1995; 1987) thinking provides a means for seeing software as another middle ground, as an immanent projection of social and machinic relations; source code and programming, rather than being a fixation of analysis, become one type of relation within a processual encounter between a problem and its conditions. Sociotechnical regime is an ongoing response to a problematic that is produced in response to a perspective on how solutions to the 'true' problem—like solving problems computationally, reliably, and predictably—are actualized. A regime is based on a plane of reference comprised of the functions it uses to affect the material world, in response to propositions from within or without its regime. Events describe a type of encounter that is noticed by someone or something; if an event is indicative of a solution—like Babbage's Analytical Engine—it becomes possible to begin to describe its associated problem. From there,

المنارة للاستشارات
www.manaraa.com

the social and machinic relations of a sociotechnical regime begin to coagulate around a plateau, which is a period that is open to expansion, and the relations a plateau captures are organized rhizomatically, which is can be truncated, pruned, and connected to any other plateau as necessary, when new insights into the nature of its relations are discovered. In this way, Hoerni's planar manufacturing method for silicon transistors expands the plateau of computation into Gutenberg's domain. And significantly, sociotechnical regimes relate to their world through a navigation of smooth and striated spaces: if a problematic encompasses the continuous, smooth conditions of an analog encounter, the solution a sociotechnical regime like electrical engineering can produce, like a transducer, will striate an analog continuum into a series of discrete values. Smooth and striated spaces allow a regime's encounter with a proposition to account for what is lost and gained in the negotiation, which also identifies a regime's principle point of contact with the world. Sociotechnical regimes conceptually describe how and why propositions are answered in the ways that they are, which allows a product like software to be seen as less than its programming or source code, and more for its processual design, development, delivery, maintenance, and abandonment.

Describing Software Engineering as a Sociotechnical Regime.

Software has always been open-ended, because it is always open to change if its source code can be accessed, modified, recompiled, and deployed, and current development methodologies, like DevOps and Continuous Delivery emphasize this perspective. As the next chapter will demonstrate, software has always been 'continuous,' because it has—generally—never entirely worked perfectly. The concept of the problem-oriented sociotechnical regime argues for an analytical emphasis that looks less at the products they produce and more on

المنارة للاستشارات
www.manaraa.com

problems which they continually engage. For software engineering, this means looking at the problem's software projects are designed to solve, and the processes mobilized to solve them. For media studies scholarship, this means devaluing the concept of 'programmer' in favor of 'software development.' Source code and software are treated as a fetish by media studies scholars because they are typically viewed as the most important part of 'software': programming is related to writing, which produces interpretations where source code is a readable, interpretable text, therefore producing narratives where the programmer is an author. This view fetishizes 'programming' by hiding and obscuring the many processes and practices, both social and machinic, that bring software to life; programming is effectively a fetish in media studies because it acts as a stand-in for the engineering and communicative practices involved in software's production. Because of this fetishism, the 'programmer' has effectively become a pastiche for scholarship examining software, which treats it as an overdetermined given. This overdetermination effectively black-boxes the many practices employed in software development, by software engineers, and perpetuates the mythologizing of the 'lone author' trope associated with the programmer identity. Engaging with problematics at the level of a regime allows programming to become one of many forms of an encounter with a problem within its conditions, which white-boxes software development by de-fetishizing the programmer, allowing software to be an attempt to solve a problem that itself emerged from a set of conditions, through a manifold of social and machinic relations, a milieu, necessitating the attempt. Once the programmer is no longer treated as a stand-in for all thing's software related, programmers and source code become less important than the problems and processes used to address problems emerging from a state of affairs.

المنارة للاستشارات

www.manaraa.com

Deleuze and Guattari's concepts, collaboratively and individually, are distinctly suited to considering software engineering because 'software engineering' is now and has historically been a set of practices and relations giving rise to a kind of software that is always becoming. A brief look at Software Development Life Cycles (SDLC) like waterfall or agile, or Software Testing Life Cycles (STLC) shows that many processes and agents feed into the development, delivery, and maintenance of software products. For example, as programmers are tasked by project managers with 'given' problems, the most effective STLC practices have testers working alongside those programmers. As Gerald D. Everett and Raymond McLeod (2007) state:

> Managers and executives of companies that develop computer software have perpetuated the myth that quality can be tested into a software product at the end of the development cycle. Quality in this context usually means software that exhibits zero defects when used by a customer. It is an expedient myth from a business planning perspective, but it ignores two truths: (1) Testing must be started as early as possible in the software development process to have the greatest positive impact on the quality of the product and (2) You can not [sic] test in quality ... period! (p. 13)

The argument Everett & McLeod make *for* testing *alongside* active development is evidenced in an axiom posited by Barry Boehm and Victor R. Basili called the "Rule of Exponentially Increasing Costs to Correct New Software" (p. 14). Essentially, before software is actively programmed, it is designed and documented. Checking for errors in the design documents before development work commences costs less than fixing an error in those documents after work has commenced. The same applies to active development: testing for errors and fixing them while a software product is being developed costs less than fixing errors after a product is delivered. Fixing errors in deployed software costs exponentially more than fixing errors during the

المنارة للاستشارات
www.manaraa.com

development process. The reasoning behind this is that errors in delivered and deployed software might only become evident after a long period of time: "the cost of defect correction must also include diagnosis at a distance, package level correction, and the delivery of the correction fixes" (p. 15). Programmers have to forensically step into a project they may have ceased working on months ago and reproduce and fix that error while hoping no new errors are produced by their fix. Working with testers can reduce the likelihood that errors will exist in the software deliverable. Even a brief look at an STLC demonstrates how a programmer's 'authorship' is only as good as the way their 'given' problem is articulated to them, and even in perfect conditions, only as good as their technical ability to solve it within the practices imparted by their sociotechnical regime. The practices encompassed by SDLC methodologies and STLC patterns attempt to reduce the likelihood that a problem posed to a project manager and their technical writing staff is misunderstood. Software development, for its continuous nature, produces software that is in the middle—kicked off by an event, spanning a plateau, forming relations to users and systems rhizomatically—while being actively developed, and lingers in a milieu even after it is abandoned, influencing other projects and design decisions decades later. The problem-oriented sociotechnical regime uses Deleuze and Guattari's concepts to capture the multiplicities of workers and practices at play in software development.

This chapter has outlined the concepts that will be used to produce a historical analysis of the events and relations leading to the creation of software engineering as a discipline that is similar to, but distinct from, Computer Science. The next chapter describes software engineering as its own problem-oriented sociotechnical regime, examining the trends leading to its formal emergence in 1968. The chapter examines the events and plateaus of early software, the formalization of Computer Science as an academic discipline, the role labor practices played in

المنارة للاستشارات

www.manaraa.com

shaping the 'software engineer' identity, and the development of software engineering as a deliberate polemic designed to formalize the problem-solving labor of computation while challenging its perceived immateriality. The production of a history of software engineering honors the assertions Deleuze and Guattari (1994) made when they stated that "new concepts must relate to our problems, to our history, and above all, to our becomings" (p. 27). The history of software engineering's inception will demonstrate how the 'programmer' and source code have long been fetishized and black-boxed by parallel practices, and how the concept of the sociotechnical regime white boxes 'programmatic authorship' and source code in favor of the orienting influences of problematics.

المنارة للاستشارات

www.manaraa.com

Chapter 3: Tracing the Emergence of Software Engineering as a Sociotechnical Regime

> [It] appears that the whole of conditions which enable a *finite* machine to make calculations of *unlimited* extent are fulfilled in the Analytical Engine. ... I have converted the infinity of space, which was required by the conditions of the problem, into the infinity of time. (Charles Babbage qtd in Randell, 1984, p. 8)

This chapter mobilizes the analytical framework proposed in Chapter 2 by interpreting 'software engineering' as a sociotechnical regime, which highlights its distinctness from its traditional enveloping by 'Computer Science.' This chapter demonstrates how a Deleuze and Guattarian examination of 'software engineering' shows it to be its own type of discipline. Software engineering resides in the thought-form of science, and necessitates a historical, processual, and intersectional analysis of software as 'solving problems computationally' to understand. As the historical account developed by this chapter will show, the principle element of distinctiveness resides in how software engineering solves problems, which a Deleuze and Guattarian theoretical framework emphasizes. For software engineering, the cares, considerations, and interpretations of problematics differ from Computer Science to such a degree that they have organized their own plane of reference by which software solutions are axiomatized and actualized. This understanding underscores the importance of thinking through software problematically by recognizing the processual ways in which it is created because it incorporates common social factors and pressures into its creation. In doing so, it provides a basis for de-fetishizing future media studies scholarship about programming and source code by placing the onus of analysis less on programmatic statements and more on the technical factors mediated by social practices leading to their actualization as software. For software programming is one practice of many, source code is one kind of text, and their meaning lies

www.manaraa.com

elsewhere, before and after the practices and epistemologies leading to its actualization. Incorporating this understanding into media studies scholarship will lead to interpretations of software as a product of equal parts, both technical and social, and will serve to de-fetishize the act of 'programming' and source code by expanding the definition of software into its many social and machinic encounters and relations, its design and implementation and maintenance.

The concept of the sociotechnical regime recognizes the explicit and reciprocating effects that problems and conditions have on each other, and how sets of axiomatics that 'solve' problems tend to cohere onto planes of references. Interpreting software engineering as a sociotechnical regime applies a Deleuze and Guattarian framework to the analysis of how a problem coheres a set of practices and identities toward the production of an idiosyncratic solution. Basically, the problems that software engineers solve typically differ from those that computer scientists solve because of their social milieus. They may use the same techniques and use the same tools—like git or Python or Microsoft Visual Studio—and even, in some cases, work on or through the same problems, but their solutions are shaped by different milieus. At the root of both Computer Science and software engineering is the enormous problem of 'solving problems computationally.' Understanding its implications involves treating the emergence of software engineering as a series of events upon a multiplicity of plateaus by tracing the historical and social factors surrounding that problem rhizomatically. Doing so furnishes a means to find the points of divergence through which 'software engineering' coalesced as a distinct identity, itself a product of a distinctly different plane of reference from Computer Science. As Chapter 2 argued, problematics—the combination of a problem with its conditions—are inexhaustible as virtualities, but are exhausted each time a solution is actualized (May, 2005). Brian Massumi (2002) uses the phrase "field of potential" to describe the kind of Bergsonian immanence which

المنارة للاستشارات
www.manaraa.com

is evident in Deleuze's thinking about problematics, e.g., "It is in becoming, absorbed in occupying its field of potential" (p. 7), which is helpful for framing the type of 'becoming' a sociotechnical regime encompasses. Sociotechnical regimes are immanent to a problematic (or a multiplicity of problematics). So, just as problems are immanent to their conditions in that they are influenced and shaped by them, problems also serve to guide and shape the nature of future conditions by determining how—in some ways—the future becoming is of things which are actualized. For Deleuze, problems are inexhaustible virtualities, or fields of potential, and solutions are exhausted the moment they are actualized, a point May (2005) helpfully elucidated; it stands to reason that different regimes have different concerns based on the nature of their social interactions within a milieu. The simple premise here is that 'software engineering,' while related to, and potentially taught by those within Computer Science, is actually a distinct discipline because of the way its conditions shape its interpretations of problems and its actualizations of solutions. If it is a mistake to conflate a software engineer with a computer scientist, it is a mistake to conflate a software engineer with a programmer: they may overlap in praxis, but only superficially. This is how media studies scholarship has fetishized 'programming' and 'source code'. By collapsing the many epistemological and ontological factors and considerations that go into software as a technic behind 'programming,' scholarship tends to treat a print out of source code as emblematic for issues of design, as a kind of Foucauldian 'author function' that fails to incorporate the significance of iterative authorship to computational work, or of the delimiting and mediating effect of a social milieu has on the 'programmer.' It is also a forest for the trees issue: sociotechnical regimes incorporate a perspective of software that sees programming in place of the needs' assessments, requirements gathering, testing, documentation, and maintenance tasks that produce software. If it is wrong to

المنارة للاستشارات
www.manaraa.com

assert that needs assessments are the entirety of software development, it is equally inappropriate to hide software behind a fetish of programming.

To understand how and why software engineering is a sociotechnical regime, and how that understanding allows media studies interpretations to properly value its problem solving and design, this chapter explores the history leading up to the 1968 NATO Conference on Software Engineering, which was a formal industrial and academic response to a period of time known as the 'software crisis' (Cambel-Kelly et al., 2014; Ensmenger, 2010; Naur & Randell, 1969, p. 70). Software engineering was and continues to be informed by circumstances and practices leading up to the 1968 NATO Conference on Software Engineering, which roughly coincided with the formalization of Computer Science as an academic discipline, e.g., the emergence of pedagogical standards released by the Association for Computing Machinery (ACM) later that year, often referred to as "Curriculum '68" (Atchison et al., 1968). The 'software crisis' narrative of the 1960s in the United States and Europe lead to tensions between academic and industrial computational efforts, between the application and suitability of theoretical knowledge, and between the management of human resources through pedagogy, corporate management structures, and an emphasis on reducing expertise to increase the predictability of software outcomes. After 1968, software engineering became the practical application of theory and disciplinary knowledge of computation through processes of management to the task of translating real-world problems into computational solutions. Through Deleuze and Guattarian problematology, this chapter will explore the negative spaces and truncated lines of flights established toward the end of the 1960s by looking at examples of how software engineering continues to resist and encapsulate aspects of management and Computer Science while behaving as a fundamentally creative and inventive field.

للاستشارات المنارة

www.manaraa.com

An effective 'Deleuze and Guattarian' history of software engineering requires us to weave four narrative threads together through the smooth and striated spaces of the 'software crisis': the trends of early computation and the establishing of a 'programmer' or 'coder' identity and role; the formalization of Computer Science as an academic discipline; the managerial impetus for wanting more authority over programmers; and the manifestation of 'software engineering' as a community of practice. Specifically, the work of this chapter examines how the nascent period of computation in the United States, and the historical narrative of the software crisis, produced a 'software engineering' community of practice that is both formally compounded and terminally unstable. Seeing programming as one practice of many will de-fetishize source code and lead to a nuanced and thoughtful understanding of how software is comprised of many parts and human relations. Source code is one kind of product software engineers produce. Once enumerated, software engineering can be seen as a sociotechnical regime with peculiar types of problems (intrinsic and extrinsic) within our contemporary domains.

Software Engineering

Software engineering's central problematics—the problems and conditions the regime continue to encounter—reside in the literature of the 1960s 'software crisis,' the NATO meeting, and the inception of the ACM's Curriculum '68. It then takes a Deleuze and Guattarian turn toward history by overlaying the three nascent periods upon current trends of Continuous Delivery and DevOps (e.g., Development and Operations), looking at how software engineering and the difference of problem definitions in Computer Science and managerial science have led to an inherent tension between abstract and practical considerations defining software

المنارة للاستشارات
www.manaraa.com

engineering as a discipline. Problems themselves will then be defined, followed by a dissection of the narrative about the work of software engineering in Frederick Brook's *The Mythical Man-Month: Essays on Software Engineering* (1995).

The software crisis of the 1960s can be understood as the intersection of plateaus and events and rhizomes that culminated in the formalizing of disciplinary lines and expertise that has not been overcome many decades later. This section produces a brief history of computation, looking at the working methods established with the inception of ENIAC and EDVAC in the 1940s. I then produce a post-war history of the emergence and formalization of Computer Science. Next, I examine the managerial practices for factors pushing against the efforts of academic computation to define 'professionalization.' Finally, I look at software engineering as a kind of *working with* computing, a solution to both the formal and practical requirements dictated by Computer Science and managerial requirements for an industry struggling with issues of 'programmer autonomy,' and the narrative tropes associated with 'black box wizardry' and unbalanced power relations which informed the eventual emergence of software engineering as a technical regime. From this discussion, we will better understand, in Chapter 4, why software engineering, as a technical regime with an unstable plane of reference, has often been conflated with and fetishized for the asignifying code (programming) it produces, rather than for the problematic apparatus it enacts and struggles through.

Early Software

> Of the many projects begun in that era, I mention only a few. I have selected these not so much because they represent a first of some kind, but because they illustrate the many different approaches to the problem. Out of these emerged a configuration that has

المنارة للاستشارات
www.manaraa.com

survived into this century through all the advances in underlying technology. (Cerruzi, 2012, p. 31)

This section examines how the trends of early computation formalized an identity and set of labor practices that persist today. Paul Cerruzi's (2012) quote above refers to a period from 1935 to 1945 where the problem of 'solving problems computationally' became feasible for engineers and scientists to axiomatize. His statement can be from a Deleuze and Guattarian perspective: plateaus began to expand, capturing the events leading to the organization and actualization of different approaches to a central problem. While many of those approaches failed, it would be wrong to say they were discarded. One of the central and binding elements of a history of computation is the influence of the 'programmer' role, which is distinct from 'computer scientist' or 'manager.' The history of early computation sheds light on the human practices and rituals entangled in relationships with computational machines. ENIAC, for two reasons, is recognized as the first modern computer in the United States: first, it was an electronic digital system; second, it could be programmed (Ensmenger, 2010). While other computational systems predated ENIAC (Williams, 2000), the significance of ENIAC (and its successor, EDVAC) for this brief history anyways, resides in how 'programming' was conceived and enacted, who performed the work, and the emergent relationships within a problematic that confounded (and defined) future expectations for 'software' and its development.

Computation with ENIAC sought to replicate a working dynamic that had been established at the U.S. Army's Ballistics Research Laboratory (BRL) during the early stages of the United States involvement in World War 2. The BRL sought to compute hundreds of tables that could improve the efficient delivery and accuracy of bombs and artillery shells, tracking

المنارة للاستشارات
www.manaraa.com

variables accounting for environmental issues, like windspeed, elevation, and humidity. The 'computation' of the tables was performed by women, who received instructions from men; from these women the first ENIAC "coders" emerged (Chun, 2005, p. 33; Ensmenger, 2010, p. 35). ENIAC was originally conceived to automate and replicate the working process employed by the 'female computers,' so when ENIAC was introduced, several female 'coders' were employed from the BRL to produce the "manual labor" required to perform the "mechanical translation" of "higher-level mathematics" into "machine language" (Ensmenger, 2010, p. 35). Programming, at this stage, was thought to be menial work: "[programmers] were former computers because they were best suited to prepare their successors: they thought like computers" (Chun, 2005, p. 33); programming, as envisioned by Goldstine and von Neumann in *Planning and Coding of Problems for an Electric Computing INstrument* was six stages of orderly and predictable process, was straightforward and rational, and if done properly, the work of a coder was simply the translation of math into machine instructions.

Problems, when expressed rationally through mathematics, were assumed to be ordered, logical, and straightforward. ENIAC was designed and implemented based on those ideals. What the male mathematicians and scientists working with ENIAC and the coders realized, however, was starkly different from 'rational' and 'orderly': not only did ENIAC not model the hand computation the women performed at the BRL, but the translation of 'problems' into electrical approximations demonstrably proved that computation was a messy business. The female coders, therefore, "would often have to revisit the underlying numerical analysis" of a problem and soon, "some scientific users left many or all six of the [Goldstine and von Neumann] programming stages to the coders" (Ensmenger, 2010, pp. 36-37). DeLanda (2010), explaining assemblage theory in Deleuze's work, described an environment where many actors interact: we can define

المنارة للاستشارات
www.manaraa.com

an assemblage in as few as two parts, be it two people interacting, or a person and a machine; or we can look at the parts of an institution, such as a municipal government, comprised of agencies. Production is an immanent product of assemblages, emerging from the interaction of their components ("legitimacy" in the case of agencies, as DeLenda argued (2010, p. 19)). Through the encounter with ENIAC, the effort of solving problems computationally gave rise to an assemblage of computer, managers, and coders, whose emergent relations expressed the difficult material realities inherent in programming.

Like any assemblage, however, the realization of ENIAC and the functional and productive use of computation during World War 2 established a working template of 'complications' as much as a new 'command' over computers, encompassing human as much as machinic elements. Problems are not only messy things but working on problems entangles and yields to a reality that is unforeseen in many circumstances. If solutions are actualizations, and problems are virtual, the solution is not only "a particular form of exhaustion" (May, 2005, p. 85), but an immanent material encounter that expresses fundamental limitations that shape the expression of the solution as much from the circumstances surrounding it as by those inherent in the problem itself. In 1965, for instance, Willis Ware stated that "[all] the programming language improvement in the world will not shorten the intellectual activity, the thinking, the analysis, that is inherent in the programming process" ("As I See It: A Guest Editorial" in Datamation, pp. 26-27); at the outset of ENIAC and its stored-program version, EDVAC, the work of *programming* was viewed simply as translation, of requiring little intellectual effort, and hence worthy of the subservient female 'computers' brought over from the Ballistic's Research Laboratory. It is a testament to those women that the grim reality of computation, the difficulties inherent in 'translation' from the virtual to the actual domains, soon became apparent: Ensmenger (2010)

المنارة للاستشارات
www.manaraa.com

states that what surprised the institutions pushing computation in the 1940s was not the revolutionary aspects of digital, machinic labor, but of how the computer programmer—the interface between the virtual problem and its actualized solution in the machine—shaped the revolution (p. 29). He explained that

> [extracting] acceptable performance and reliability out of the early electronic computers required an enormous degree of messy tinkering, local knowledge, and idiosyncratic technique. (p. 31)

'Computer' problems, conceived in their inexhaustible virtuality, were truly virtual. The 6-staged abstracted set of practices of Goldstine and von Neumann were also virtual. For ENIAC, the emergent properties of successful programming were immanent to the relationship of male managers and scientists (who were supposed to complete the first five stages) ordering female subordinates to 'simply translate' the mathematics of a problem into machine instructions that were recognizable as a solution for that problem. But as the difficulties inherent in the actualized material realities of the solutions emerged, in the form of running programs, the relationship of 'master' and 'servant' inherent to the command-nature of computation became subverted when the female 'coders' often completed all six Goldstine and von Neumann stages themselves (Ensmenger, 2010; Cerruzi, 2012). Essentially, through exposure to the machine and the production of 'local knowledge' and 'idiosyncratic technique,' the female coders discovered and subverted the limits of a problem's virtual imposition upon the actual within the assemblage of masters, servants, and machines. The programmers—those women coders at the outset of the United States efforts to leverage computation—not only succeeded but set the stage for the digital revolution.

www.manaraa.com

Early programmers ('coders') were problem solvers as much as they were problem translators, but the autonomy they required to become *effective* produced systemic subversions within the assemblages within which they were employed. At the center of that stage resided a new power dynamic that required autonomy to produce expertise, i.e. time to perform 'messy tinkering' to learn and to become effective at translating virtual problems into actualized solutions: "[until] about 1950, it was a major accomplishment if one could get an electronic computer to operate without error for even a few hours" (Ceruzzi, 2012, p. 29). Not only were the machines difficult to operate and maintain, but the reality of the autonomy required to learn how to keep them running, however, was tempered with the fact that the programs were error-prone. As Maurice Wilkes put it, himself an author of arguably the first book on programming in 1951 and a mathematical physicist, "a good part of the remainder of my life was going to be spent in finding errors in my own programs" (qtd. in Cambell-Kelly, 1992, p. 22). If effectiveness is measured by the efficacy of the solution, errors—"bugs" or 'moths in the machine' (Kohanksi, 2000)—would eventually undermine the profession, because those errors were inherent to the work of translating continuous values and processual becomings into digital, discretely manifested approximations.

This section described how the role defined for early programmers established both an identity and labor model that would become formalized through academic and managerial efforts. The significance of this for a Deleuze and Guattarian history, and for the sociotechnical regime of software engineering, resides in how the identity and labor role connected to, disrupted, shaped, and were resisted by the thought forms attempting to define it. Expertise became a line of flight for programmers through the early history of computation within highly structured academic and corporate environments. The issue persisted into the period known as

المنارة للاستشارات
www.manaraa.com

the software crisis; the expertise programmers needed to perform their work is precisely what would become resisted in many organizational ways, because software developers were idealized as passive receivers, translators, rather than as active thinkers and agents of change. The next section describes the organization of Computer Science.

1968, Formalization of Computer Science

This section describes how Computer Science organized itself as a plane of reference and thought form by highlighting some of the difficulties early computationalists experienced as they struggled to form their own identity within American and European academic environments. Computer Science (CS) was formed as an academic discipline to, in an academic context, legitimize the study of theoretical computation, and was a response to cultural pressures to professionalize, to become authoritative in the creation and application of software to problems both actual and virtual. In 1968, Atchison et al. worked to both formally name the discipline of 'Computer Science,' and outline its areas of study. The report, published in Communications of the ACM, expressed Computer Science as the conjunction of three major areas of study: information structures and processes; information processing systems; and methodologies (pp. 154-155). This division of research areas with their associated subtasks—data structures and programming languages are enfolded into the information structures and processes category—not only defined Computer Science then but continues to define modern Computer Science departments now. Significantly, Curriculum '68 is not only a singularity in the Deleuze and Guattarian sense but is rather an ongoing event that continues to become by continuing to reside over and interact with its own plane of reference and parallel plateaus of managerial science and software engineering. The curriculum defined by Atchison et al. (1968), like any specification,

المنارة للاستشارات
www.manaraa.com

defines what is and more importantly, what is not. It allows certain connections to grow rhizomatically while trimming others, precluding lines of flight. Computer Science, as it formalized and legitimated itself institutionally within academic contexts during its formative years, excluded instruction in praxis, or the pragmatic application of programming for practical purposes. And in fact, by cementing the gap between the (cliched) divide of theory and practice, CS legitimated the study of computation as an intellectual activity worthy of scientific investigation.

In the late 1940s and 1950s in the United States and Europe (notwithstanding the Soviet Union and other understudied sites of historical focus), computation, which had first served at the pleasure of World War 2, was primarily subordinated to several academic disciplines and industries. Universities, like Harvard or Princeton, established computing departments that served *other* departments; the study of computation itself, and the standard unit of analysis—the algorithm—would not be established until the 1960s. Computation, initially, was a service. Those who worked with and specialized in computation and programming were individuals from many backgrounds—chemistry, physics, mathematics, economics—who took an interest in the work, but found themselves—as a product of that interest—subordinated to the political and social pressures of institutions attempting to locate and task computational resources appropriately (Ensmenger, 2010; Cambell-Kelly et al., 2014). The problem was of realizing the value of computation. But as their experiences *with* computation grew, the engineers and scientists tasked with working on other people's problems, began to structure a way of seeing and knowing that used the computer as a lens for the world through a principle of generalization through algorithmic thinking: "Programmers too often saw their work as temporary solutions to local problems, rather than as an opportunity to develop a more permanent body of knowledge

المنارة للاستشارات
www.manaraa.com

and technique" (Ensmenger, 2010, p. 112). The formalization of what would come to be known as Computer Science (several terms competed for the distinction) is an example of how a Deleuze and Guattarian plane of reference, as an axiomatized set of attitudes, behaviors, and techniques, comes to produce a technical regime. What makes the instantiation of Computer Science interesting, however, is the way its formalization as an *episteme* demonstrates Deleuze and Guattari's assertion that the sciences work to principally *slow down* the circumstances of an event, such that the resulting plane of reference becomes the sieve by which chaos is filtered. Computer Science emerged as one solution to the inexhaustible and chaotic problem of capturing, filtering, and approximating often analog states of affairs in digitally discrete logic and processes.

'Computation in general' distinguishes itself from other machines in its generality, and is the ultimate form of human mediation, for it sits astride the virtual chaos of the problem and the filtered, and desirable predictability the data that it manipulates, its manipulation is the solution. A computer's programmability is its ability to interface with the world, to output work based on something input; this programmability, the reality that a computer can be interfaced with other hardware or software, allows it to function as a machine-of-machines, a controller issuing commands to other machines in a network of relationships. Computers sit at the heart of many of today's assemblages, and the plateau of Computer Science has expanded to produce an orientation toward the world that integrates those controllers into the general problems of human affairs, evident from the basic timeline of its evolution depicted in Table 3.1.

المنارة للاستشارات
www.manaraa.com

Table 3.1: Significant Events in the Inception of Computer Science

Event	Where	Date
Charles Babbage began designing the Difference Engine to compute the values of polynomial functions	England	1822
Charles Babbage described the Analytical Engine, the first general-purpose computer	England	1837
IBM donated tabulating equipment	Columbia	1920
Moore School of Electrical Engineering started	UPENN	1923
Vannevar Bush created Differential Analyzer	MIT	1931
IBM established Thomas J. Watson Astronomical Computing Bureau	Columbia	1934
Samuel Caldwell is teaching 'graduate seminar in machine computation'	MIT	1935
Alan Turing describes a universal computer for solving any describable problem	England	1936
Konrad Zuse realizes that data and code are the same thing to a general purpose computer	Germany	1937
Began construction of ENIAC at Moore School	UPENN	1943
IBM built the Harvard Mark I	Harvard	1944
EDVAC, the first stored program computer, was conceived and proposed to the Ballistics Research Laboratory	UPENN	1944
IBM's Columbia Computing Bureau became the Watson Scientific Computing Laboratory	Columbia	1945
First computer course taught at Moore School	UPENN	1946
Completed construction of ENIAC at Moore School	UPENN	1946
Established Master's degree program in Mathematics with focus on 'computing machinery'	Harvard	1947
Established PhD program with computing focus	Harvard	1948
EDVAC was delivered to the Ballistics Research Laboratory	Eckert-Mauchly Computer Corporation (EMCC)	1949
Project Whirlwind became the Digital Computing Laboratory	MIT	1951
Louis Fein rigorously defined 'Computer Science' as an academic discipline	Communications of the ACM journal	1959
Edsger Dijkstra completed his dissertation about 'communication with an automatic computer'	University of Amerstdam	1959
C. M. Sidlo argued for scientific professionalization of computational studies	Communications of the ACM journal	1961
Formed Computer Science program outside of the Computational Laboratory	Harvard	1962
Anthony Oettinger described 'Computer Science' as a hodgepodge of "bits and pieces from other disciplines"	Presidential Letter to the ACM Leadership	1966
Offered undergraduate degree in Computer Science	MIT	1969
Computer Science became a formalized program (inside of Electrical Engineering)	MIT	1969
Malcolm Gotterer argued for theoretical standardization	Proceedings of 1971 ACM Annual Conference	1971
Project MAC became the MIT Laboratory for Computer Science	MIT	1975
Offered graduate degree in Computer Science	MIT	1979

The lineage of the modern computer was glimpsed first in the 19th century by Charles Babbage, who, as Cerruzi (2012) notes, "[anticipated] the notion of the universality of a programmable machine" (p. 28). In contemporary terms, the 'universality of a programmable machine' is now ubiquitous; its universality is inescapable, from embedded microcontrollers in wearable internet-of-things (IoT) devices and smartphones, to integrated Central Processing

المنارة للاستشارات
www.manaraa.com

Units (CPUs) on desktop and laptop computers: the data of our lives is squared around its edges, conforming to the approximations of the digital limitations inherent in the discrete nature of the machinic mediator. In 1937 Germany prior to World War 2, Konrad Zuse realized "that the operations of calculation, storage, control, and transmission of information, … were in fact one and the same" (Ceruzzi, 2012, p. 24). A year earlier, along parallel lines of thought, Alan Turing published work that defined the concept of the universal machine in response to a problem German mathematician David Hilbert posed. The idea of the universal machine is one of sufficient generalization, where its ability to enact general purpose instructions allows it to respond to and act on descriptions of human problems, if those descriptions are formalized in some way. Between the idea of a universal machine, and the realization that code (calculation, control) is the equivalent to data (storage), a plane of reference yields its organization to a systematic axiomatization of reality, and the principles which determine how the continuous curves of a state of affairs are squared, are made to fit discrete, digitally abstracted containers. What is universal in the machine Turing envisioned is its approximating nature and the strength of its mediation upon the data it inputs and outputs. But as a testament to the importance of 'lineage,' and how significant insights tend to build on the work of others, Alonzo Church published Church's Thesis in 1935, which still impacts computational theory today, according to Adam Olszewski et al. (2006). Church's thesis deals with the nature of "partial recursive functions," and the assertion that "there is no recursive decision procedure for first-order logical validity" (Mendelson, pp. 228-229), which turns out to be analogous with Turin's work on what are now called "Turing-computable" functions (p. 229), but is arguably unprovable because it is a "rational reconstruction" rather than a scientific proposition with its related functions. According to Elliot Mendelson, Turing effectively proved Church's thesis, which shows how

المنارة للاستشارات
www.manaraa.com

ideas germinate and bear fruit—even ideas which at first are refuted for not being scientific enough.

Before its material emergence during World War 2, computation and some of its potential applications was reasonably well understood, from a theoretical perspective. Zuse, a mechanical engineer by training, and Turing, a mathematician, were inspired by the problems they were seeking to overcome and solve. The theoretical basis of what a computer should be—from Turing's universal machine to the brain-like von Neuman architecture of memory, register, and stored programs—produced, "in retrospect," Ensmenger (2010) argued, an "almost overdetermined" creation of "an academic discipline devoted to Computer Science" (p. 115). Computers, and those who worked with them, were invariably interdisciplinary at their formalization in ENIAC and EDVAC, and the instantiation of the first computers had been foreseen for quite some time before the first systems in England and the United States went live. But the creation of a science devoted to computation had its roots—like any technical regime—in self-interested and externally motivating factors. While authors like Ensmenger (2010), Cambell-Kelly (2014), and Ceruzzi (2012) outline the broad strokes of the cultural and societal impetus to 'professionalize,' Robert Baber's (in Glass, 1998) personal account of learning electronics and programming during the late 1950s highlights a gap between theory and practice that informs the basis of the diverging plateaus of Computer Science and software engineering:

> Because of my education, I was accustomed to basing designs on theoretical and mathematical models of the artifact being designed. However, in my experience, writing programs, both as a student engineering in programming groups and later at the IBM 1401 installation, seemed to require just the opposite. No mathematical models existed for programs or for designing programs. Obviously, programs had mathematical, logical

المنارة للاستشارات
www.manaraa.com

> aspects, but the theoretical-mathematical foundation for programming as a whole was missing. … The commonly observed high error rate in the programming activity clearly made such a mathematical foundation for programming … desirable, since it would presumably lead to design error rates comparable to those in the engineering fields, that is, much lower. (pp. 180-181)

Three salient points are evident in Baber's account: he worked from and adhered to the plane of reference he had been educated in; he experienced a disconnect between the application of the theory to the practice of programming; the actualization of theory through programming was error-prone principally because, "as a whole," the "theoretical-mathematical" plane had issues interfacing with computational machines.

Theory and praxis continue to be a line of division between software engineering and Computer Science. Historically, Computer Science does not *necessarily* focus on programming and its inherent difficulties; programming emerged into the world through haphazard arrangements of immanent expressions, was difficult to learn, difficult to enact, and difficult to teach. So, while programming is at the root of Computer Sciences' expression, in the sense that what programmers produce—code—is a kind of transduction[1] of a selection set of axiomatic principles into a machinic, asignifying language, Computer Science, as a discipline, chose to legitimate itself academically by focusing on the 'theoretical-mathematical' plane rather than through the praxis of programming. As the women ENIAC coders well knew, programming at its outset was distinctly *not* mathematical: at least, not purely mathematical. Programming was

[1] Guattari (2014) explained that "[process], which I oppose here to system or to structure, strives to capture existence in the very act of its constitution, definition and deterritorialization. This process of 'fixing-into-being' relates onto to expressive subsets that have broken out of their totalizing frame and have begun to work on their own account, overcoming their referential sets and manifesting themselves as their own existential indices, processual lines of flight" (p. 29).

المنارة للاستشارات
www.manaraa.com

frustratingly *idiosyncratic*: Betty Jean Jennings, one of the six original ENIAC programmers, noted that "[since] we knew both the application and the machine, we learned to diagnose troubles as well as, if not better than, the engineers" (qtd in Ensmenger, 2010, p. 37). At the outset, there was no sufficient theoretical-mathematical basis for transducing the problems handed to them by the male engineers and scientists at the Ballistics Research Laboratory, so the women charted their own plane of reference which integrated both the axiomatic principles of the scientific and engineering fields they were translating, and the immanent and emergent principles of ENIAC and its local idiosyncrasies.

As transducers, while programmers lived astride multiple planes of reference which required pragmatic selection and application of techniques, the basis of their pragmatism was problematically not a tenet of contemporary education, but rather of self-learning. Aspects of abstract theory, like those formalized by ACM's Curriculum '68, or referred to by Baber (1998), were unquestionably helpful, as they improved the ability for a programmer to provide more appropriate techniques to the implementation of a solution from a problem, but the selection and *real* application of theory involved moving through a virtual space of possibilities and potentials by emerging into an actual space of material immanence (Deleuze & Guattari, 1977; 1987). What was found from the initial encounters with ENIAC, EDVAC, and its successors, was that only practical experience could shape individuals into effective programmers. Computer science's attempts to legitimize itself academically would preclude, by and large, the types of practical experiential factors that produced effective programmers. In 1959, Price Waterhouse—today a massive services company employing, at the time of this writing, over 236,000 people—published a report entitled, "Business Experience with Electronic Computers" which described the domains of knowledge competent programmers should have: "systems analysis and design is

المنارة للاستشارات

www.manaraa.com

as important to a programmer as training in machine coding techniques; it may well become increasingly important as systems get more complex and coding becomes more automatic" (Conway et al., p. 81). As per ACM's Curriculum '68, Atchison et al. (1968) recognized that the Curriculum Committee's "recommendations are not directed to the training of computer operators, coders, and other service personnel," explaining that "[training] for such positions, as well as for many programming positions, can probably be supplied best by applied technology programs, vocational institutes, or junior colleges" (p. 154). Programmers were—and arguably still are—viewed as service personnel based on the amount of instruction modern CS curriculums devote to issues one might expect to encounter in a corporate environment. Evidence of this assertion can be found by comparing the map in Figure 3.1 of courses Atchison et al. recommended in Curriculum '68 to a modern Computer Science course catalog like the one at North Carolina State Universities (NCSU's) curriculum seen in Figure 3.2. Practical instruction, teaching a student to become a professional engineering in a team environment, for all intents and purposes encapsulated by the term 'software engineering,' was not part of the original map in Figure 3.1, but has been incorporated to some extent in NCSU's catalog by way of three classes. The first, CSC 326, is a software engineering course for undergraduates that attempts to teach the

> [application] of software engineering methods to develop complex products, including the following skills: quality assurance, project management, requirements analysis, specifications, design, development, testing, production, maintenance, security, privacy, configuration management, build systems, communication, and teaming. (Computer Science, NCSU, http://catalog.ncsu.edu/undergraduate/coursedescriptions/csc/)

www.manaraa.com

Much of what the software engineering course attempts to impart are communication skills. The other courses offered focusing on practical instruction of principles of the kind Price Waterhouse wanted are CSC 510, also called "Software Engineering," and CSC 519, which focuses on DevOps, an approach to so-called "Modern Software Engineering Practices" integrating systems' support and development practices continuous delivery project lifecycles. As of the time of this writing, 1 of 48 undergraduate course offerings explicitly focus on business needs like "quality assurance" and "requirements analysis"; 2 of 69 graduate course offerings (excluding CSC 800 level seminar and doctoral research and examination listings) focus on the practical, communicative aspects of engineering.

المنارة للاستشارات

www.manaraa.com

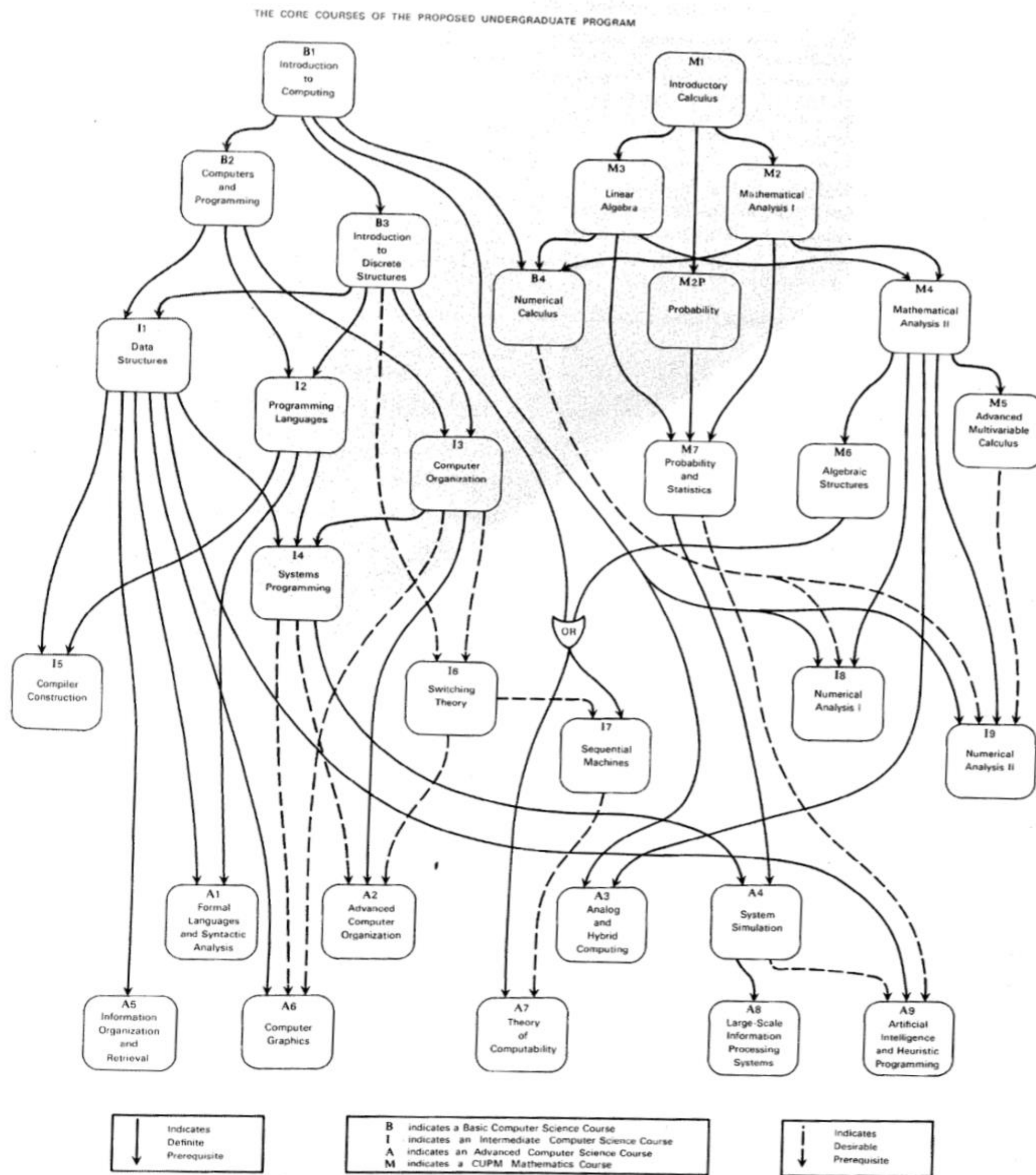

Fig. 1. Prerequisite structure of courses

Volume 11 / Number 3 / March, 1968 Communications of the ACM 157

Figure 3.1: The Curriculum Map of Math (“M”), Basic (“B”), Intermediate (“I”), and Advanced (“A”) courses established by Atchison et al. in 1968.

It is safe to take NCSU’s Computer Science department as emblematic of Atchison et al.’s Curriculum ’68 both in how it represents the majority of American university curriculums,

المنارة للاستشارات
www.manaraa.com

and for the lack of divergence it demonstrates from the norms established by the committee's work toward the end of the 1970s. "Systems analysis and design," to borrow from Price Waterhouse's report, incorporates the perspective of human and computational orientations: what is talked about between business units in a corporation, for instance, must be translated in some way into an operational system or tool.

Despite the lack of praxis-based education, the legitimizing work of those in the ACM of establishing Computer Science as an academic discipline and scientific avenue of study produced an obvious 'expertise' leadership role in the labor market. Corporations in the United States and elsewhere looked to Computer Science graduates as a source of expert-labor. Cambell-Kelly et al. (2014) note that while ACM's 1968 curriculum "helped solve some critical problems for the software industry" by establishing "the theoretical foundations of the discipline," Computer Science departments "could not … produce enough programmers to satisfy the demands of the larger commercial computing industry" (p. 184). The reasons for this shortage were that "[university] degree programs took too long, cost too much, and excluded too many people (including … many women and minorities)": demand outpaced supply, and the pursuit of academic *bona fides* produced a disciplinary culture which, from the perspective of an IBM recruiter during the 1960s, failed to "pay serious time and attention to the applied work necessary to educate programmers and systems analysts for the real world," according to Cambell-Kelly et al. If we take as true what Deleuze and Guattari (1977) stated, that "[if] desire produces, its product is real" and "can be productive only in the real world" (p. 26), then the desiring-subject of formative Computer Science, the becoming of the discipline which at first was "missing in desire," controlled the nature of its connections to other disciplines (or 'machines' in a Deleuze and Guattarian sense) to the extent that it excluded stakeholders from its nascent inception and

www.manaraa.com

resulting organization. Computer science had to disconnect from corporate influencers to both legitimize its focus on theory and obtain autonomy from those, like IBM or Remington Rand, who steered the production and adoption of computing standards after World War 2. While Wing (2006) argued that Computer Science "inherently draws on engineering thinking, given that we build systems that interact with the real world" (p. 35), the praxis and theory gap was then, as it is now, a product of design.

Even a brief comparison of ACM's Curricilum '68 map and NCSU's CSC course catalog offerings demonstrate how 'software engineering' or "engineering thinking" as Wing stated seems to be a byproduct rather than aim of Computer Science's promulgation. In 1966, Anthony A. Oettinger, then president of the ACM, a letter which discussed the close relationship between science and engineering:

> A concern with the science of computing and information processing, while undeniably of the utmost importance and an historic root of our organization [i.e. the ACM – BM] is, alone, too exclusive. While much of what we do is or has its root in not only computer and information science, but also many older and better defined sciences, even more is not at all scientific but of a professional and engineering nature. We must recognize ourselves—not necessarily all of us and not necessarily any one of us all the time—as members of an engineering profession, be it hardware engineering or software engineering, a profession without artificial and irrelevant boundaries like that between 'scientific' and 'business' applications. (qtd in Meyer, 2013)

Oettinger's letter in *Communications of the ACM* is interesting for three reasons. First, as Meyer (2013) points out, Oettinger's use of the term 'software engineering' predates the "eponymous 1968 NATO conference." Second, Oettinger recognized 'Computer Science' to be, in a Deleuze

www.manaraa.com

and Guattarian way, a plane of reference comprised of the intersections of other planes: calling it anything but 'professional and engineering' was 'artificial.' Third, his letter demonstrated that there was resistance to the exclusion of praxis—engineering—and 'business' (and its needs) from the theoretical pursuits of computing and information science. Oettinger knew that the study of computation required engineering and business practices and was warning against the "irrelevant" disciplinary "boundaries" that were taking hold in the culture of the ACM as it sought to legitimize itself academically, and his "letter to the ACM membership" mirrors, in some important ways, the arguments implicit and explicit to Wing's (2006), 40 years later. A peg of Wing's argument, after all, was that computational thinking not only "[complements] and combines mathematical and engineering thinking," but is "[a] way that humans, not computers, think" (pp. 34-35). Oettinger and Wing both recognize, explicitly or implicitly, connections of the so-called 'real world' to the abstract study of algorithms and the science of computation. For, despite the efforts exerted to exclude corporate (or 'neoliberal') influence over matters of curriculum and training in Computer Science, principles of management science, from corporate sponsors and researchers, have altered and shaped the plane of reference upon which the plateau of Computer Science draws, due to the ways in which technical regimes are internally self-interested, and externally motivated.

This section described how Computer Science came to be its own thought form by organizing a plane of reference that was both similar to and distinct from the axiomatics its early practitioners employed. Computer Science is a discipline borne of interdisciplinarity. The role of the 'software developer,' or of programming itself, falls into a theory and praxis duality: the emphasis of Curriculum '68 is on theoretical computation, rather than the broad array of engineering practices required to develop reliable software systems; the software developer, as a

المنارة للاستشارات
www.manaraa.com

functional laborer, is seen as a resource falling under managerial praxis. The next section describes how corporations dealt with issues of expertise, underscoring their influence on the founding discussions that would take place during the NATO Conference on Software Engineering in 1968, an important year for histories of computation.

The move toward de-fetishizing ‘programming’ and ‘source code’ for media studies involves locating the practice of programming in its broad historical context: programming in Computer Science is a means to an end. While *programming languages* and compilers are an area of study falling under the purview of Curriculum ’68, the category is aligned with Data Structures and Models of Computation, a subset categorized under “Information Structures and Processes” which is one of the five areas defined at the time (Atchison et al., 1968, p. 154). Programming *languages* are at least as important as data structures and models of computation. The other major areas of research defined in Curriculum ’68 are Information Processing Systems; Methodologies, Mathematical Sciences, and Physical and Engineering Sciences. Looking closely at Physical and Engineering Sciences, one can safely categorize ‘programming’—the act of producing source code—under the 7th category of Physical and Engineering Sciences, i.e., “Coding and Information Theory.” Placed alongside categories involving the production of circuits, heat management, theories of communication and control, programming is an area of research that, while important in Computer Science, is not in and of itself the disciplinary emphasis. If programming is not the most important area of investigation for Computer Science, the reasons for fetishizing source code and programmers begin to evaporate. De-fetishization invites scholarship in what ‘Coding and Information Theory’ might mean in practice.

المنارة للاستشارات
www.manaraa.com

Software in Crisis: Intersections of Computation and Management Practices

This section explains the managerial impetus for wanting more authority over programmers by controlling the nature of their expertise. Managerial praxis, as its own thought form, is a plateau encompassing many events over grand periods of time; this section simply looks at historical evidence from the perspective of a corporate community of practice in and around the software crisis. From a managerial perspective, there are four crucial factors that come into play about the 'software crisis' that impacted, on one hand, the legitimizing mechanism of Computer Science as an industrial power, and on the other, the creation of software engineering: first, the crisis was fundamentally about costs and predictability; second, the crisis was about expertise and reliability; third, the crisis was in part due to how difficult it was to define 'professionalization' for programmers; and fourth, the culture of corporate middle-management resisted the interdisciplinary expertise inculcated by programmers doing, as McCracken (1961) called it, "systems work" (p. 9) principally because such work eroded middle-management's authority and power. As the communities of practice that would comprise 'Computer Science' sought legitimacy as an academic discipline, managerial practitioners—like those of Price Waterhouse, or any business attempting to leverage computing for competitive advantage—attempted to make the most of computing resources through the exertion and maintenance of authority to control the *human* costs associated with the disruptive, emerging technologies. While many skills informing professional praxis can be taught at trade schools or universities, the quality of programming differs in almost logarithmic shades, such that its effectiveness could be measured in relation to the location of the solution along a problem's difficulty curve. But difficulty is not univariable; it is a matrix of variables and their relations and human factors and material, machinic affordances. If source code, as Chun (2008) contends, is

المنارة للاستشارات
www.manaraa.com

“the ultimate performative utterance” according to those in software studies, the recognition of managerial practitioners during the software crisis demonstrably showed that code was only important in so far as “its effectiveness depends on a whole imagined network of machines and humans” (p. 299), which requires both a metric of ‘effectiveness,’ and the ability to measure the product of automated and human coders against their actual, tasked problems. Producing reliable ‘programmer labor’ was (and remains) difficult if the historically demonstrable reputation of ‘software as unreliable’ maintains its status quo. The software crisis, from a managerial perspective, was both a recognition that traditional routes to professionalization—like curriculums in trade schools and universities or certifications—was not producing efficacious results, and the realization that the full extent of the problem had no equally ‘full’ solution. Issues of expertise reside at the heart of the (ongoing) software crisis.

First, the software crisis was largely a realization that, while business and academic expenditures on computational resources and expertise had grown exponentially from the time the first machines from IBM and Remington Rand (among others) made it into the market place after World War 2, the return on investment businesses, defense contractors, and universities received were poor: not only was the hardware expensive and difficult to maintain, but the cost of the expertise required to develop and maintain the software and hardware for those installations far outpaced the initial hardware expenditures. Thus, while attempts to automate business practices like human resources or payroll *required* rhizomatic, emergent encounters within the states of affairs new (and experienced) programmers found themselves, it took an unpredictable amount of time before the programmer could be considered ‘competent’ within that business, i.e., capable of producing *efficacious* code for solutions to problems posed by management. The problem was both extrinsic and intrinsic to the business environment for

المنارة للاستشارات
www.manaraa.com

several reasons. As Ensmenger (2010) noted, "[the] problem was a familiar one for the industry: although most employers … believed that only 'competent' programmers could develop quality software, no one agreed on what knowledge and abilities constituted that 'competence'" (p. 186). We like to think of university curriculums teaching students how to write Java, Python, C++, or any number of standardized, automated, and cross-platform languages, but early programmers typically had to learn to directly code a machine (in the singular sense) using byte code or an assembler (specific to that machine), which they may have had no access, and therefore no ability to learn to code for, the machine they would be working with prior to being hired for their position. Programming knowledge was thus intrinsic and institutional, and as such, external factors, like the availability of computing resources, tools, and programming classes at their university would have only prepared a programmer in a general sense. "Standardized" languages that provided a basis for generalizable expertise were not released until 1957 (IBM's FORTRAN) and 1959 (the U.S. Government's COBOL), and committees like the 1962 RAND Symposium, debated whether those languages were 'standardized' or merely 'common' due to differences between compilers on different mainframes. Both corporate managers and technical experts "were concerned with the apparently inability of existing software development methods to produce cost-effective and reliable commercial applications" (Ensmenger, 2010, p. 141). As the software crisis evolved and grew in the United States, the only sure-thing about 'solving problems computationally' was that it was reliably expensive.

Second, because expertise was *relative* to each business, being both expensive to inculcate and difficult to keep, programmers earned a reputation for 'unmanageability,' which coincides with software's inherent 'unreliability' and error-prone nature. When ACM's Curriculum '68 formally abdicated issues of 'corporate' or 'neoliberal' praxis in the education of

المنارة للاستشارات
www.manaraa.com

computer scientists, it left the education of business programmers and data analysts to corporations. Corporate pedagogies, for lack of a better term, essentially ensured that corporate and government sponsored languages would dominate the working-programmer's day-to-day existence; this would, in turn, feed back into and influence the tools and languages that universities would adopt. For example, Harold Joseph Highland (in Glass, 1998) was a student in the Graduate Business School at Long Island University in 1958 when the first computer was installed at the university, an IBM 1620; the system came with the newly-released FORTRAN, a language developed by IBM to solve scientific and engineering problems using a mathematical syntax. FORTRAN is short for "FORmula TRANslation," and John Brackus' team at IBM hoped the math-like syntax of the language would improve the ability of programmers to both learn programming and write reliable software. As a language, while FORTRAN was not the first to use a compiler, which is a tool that automates the translation of programming statements into machine instructions, it was the first 'automated programming language' to produce high-performing code in a timely manner. Earlier attempts at automation, like the FLOW-MATIC programming language and tools championed by Grace Hopper, produced laboriously slow results and were untenable as candidates for industry standardization. Brackus' team, with FORTRAN, produced a language and toolset which not only reduced the expertise required to program a machine, but simplified programming and increased software reliability generally. The language specifications and compilers of FORTRAN and COBOL (in 1959) allowed—and still allow—programmers to interact with a computer without having to know the underlying machine instruction set or assembly language, which has come to be known as a higher (if not 'high') level language. FORTRAN and COBOL not only reduced the amount of idiosyncratic and specific expertise required to interact and work with a computer, but FORTRAN allowed

المنارة للاستشارات
www.manaraa.com

programmers to use algebraic statements to make their mainframes perform work and COBOL's compiler translated English-like syntax into machine instructions.

Third, 'programmer professionalization' was an emergent property from a set of a relations that continually changed throughout the software crisis. While compilers reduced the expertise required to learn how to program the underlying machine, languages like FORTRAN arguably *did not* reduce the difficulty of programming effective solutions in both business and academic environments, based on the sweeping complaints driving the "software crisis" narrative which came to pass in the 1960s, hence McCracken's 'systems work' statement. Programmers were being called upon to touch many aspects of a businesses operating environment to implement systems that captured and met their employer's requirements. So, until standardization was introduced, compilers differed in important ways across mainframes (or operating systems and platforms, today), meaning work performed in one version might not translate to another version on a different mainframe. The first effort to standardize FORTRAN occurred in 1966 (i.e., FORTRAN 66), where a specification was established and recognized by the American National Standards Institute (ANSI) such that any FORTRAN 66 compiler—to be recognized as such—would meet the requirements established by that specification. Standardization was an attempt to *reduce* the extent to which idiosyncratic expertise was required to build, deploy, and maintain software in corporate environments.

Fourth, the expertise required to efficaciously use computers to solve problems disrupted traditional modes of authority in corporate environments. Corporations in the 1950s and 60s could easily purchase computer hardware, but had a difficult time making it work well, in applying its capabilities in ways that reduced costs and simplified or solved problems. So, it is an understatement to say that the expertise required to use computational resources—to conceive of

المنارة للاستشارات
www.manaraa.com

and implement software—was disruptive. On one hand, programming expertise was not only difficult to learn (and teach) but was domain-specific and difficult to generalize across corporate environments. As Cambell-Kelly et al. (2014) note, in the 1950s "it became almost a tradition that newcomers to programming simply could not be told" what they needed to know to do their jobs, and that almost universally programmers "had to learn the truth about debugging in the same painfully slow and expensive way," which continues into contemporary times (p. 169). Programmers often learned how to program a specific machine; upon transitioning to another position at a different company, often restarting the learning process. But on the other hand, and significantly more important in terms of authority, programmers in early corporate environments touched broad swaths of those environments—the "systems work" which McCracken (1961) detailed included tasks like requirements gathering and needs analysis, which meant talking to those looking to benefit from the product being developed ('stakeholders'), designing and diagramming the solution, to its programming and implementation. And then, systems work included tasks like testing and ongoing maintenance, to answer questions like, "does the program do what it claims to do?" Because the process of bringing software into life was difficult and involved, Brooks' (1995) observation is particularly salient: "[for] the human makers of things, the incompletenesses and inconsistencies of our ideas become clear only during implementation" (p. 15). Managers expected software *products* (final things) but received *processes* (ongoing things). The realization was that *programs are rarely ever finished*, and as such, programmers disrupted middle-management authority structures by becoming *indispensable*. To combat this, programmer expertise had to be reduced.

The rhetoric of the 'software crisis' of the 1950s and 60s is the backdrop to an intersection of plateaus from which 'software engineering' emerged, and the event of

المنارة للاستشارات
www.manaraa.com

intersection between Computer Science and managerial plateaus continues to define and shape expectations for software engineering. The way Computer Science organized itself after Curriculum '68 established a 'praxis' and 'theory' divide demarcating the theoretical, academic concerns of a scientific discipline from mundane business or corporate realities. Curriculum '68, as an artefact of ACM effort, was notable for its parallelism in purpose in that it not only formalized a discipline, but by means of definition and by organization of a plane of reference defined the broad notion of 'professionalization' of computer-oriented work by *resisting* and undermining the efforts of other groups from defining the term. The ACM actively worked against the Data Processing Management Association (DPMA)'s effort to offer a Certificate in Data Processing (CDP) program in the 1960s (Cambell-Kelly et al., 2014, p. 184); the definition of what constituted a 'professional programmer' for the ACM was about domination over a nascent and burgeoning industry, and for its curriculum to be valued above its competitors, it had to discredit those competitors' offerings. Yet, Computer Science, despite eventually becoming *de facto* in terms of credibility, *did not* (and arguably could not) meet the needs of business during the software crisis, which continues to both be well-regarded and insufficient today (c.f. The Next Web, 2016, "Why half of developers don't have a computer science degree"). As Ensmenger (2010) explains, "[perhaps] the most important reason why the 'personnel problem' dominated the industry literature during the late 1950s and early 1960s has to do with a fundamental structural change in the nature of software development" (p. 20): human relationships with computers and the ways computers are used to solve problems have always been changing. During the post-war period, the rate at which computational hardware improved outpaced the tools and techniques necessary to take advantage of the hardware, and a

المنارة للاستشارات
www.manaraa.com

consequence of this gap (which has not been solved in modern contexts, either) is a workforce that must continually re-educate itself as new tools and techniques promise to close it.

While the ‘programmer expertise problem’ was viewed differently by its stakeholders, its crux resided on the fact that, as McCracken stated, real-world programming requires “not so much a body of factual knowledge” but of “problem solving” skills, of which “no one knows how to teach” (qtd in Ensmenger, 2010, p. 21). Essentially, neither curriculum—that proposed by the ACM in 1968, or of the arbitrary requirements listed for the DPMA’s CDP—covered the breadth of knowledge required to instill programming expertise in formal ways. Organizations, from a corporate management perspective, were at risk of “being held hostage by their ‘whiz kid’ technologists” due to their inscrutable, inculcated expertise (Cambell-Kelly et al., 2014, p. 183), and further, the expertise came at a high cost and produced unreliable results. From a Deleuzian standpoint, his focus on the movement of problems through virtual and actualized domains and his emphasis on the ‘orientating’ role of problems sets up a conversation about the how plateaus and their intersections demonstrate the concrete ways in which problems ‘become real,’ and are at the same time inexhaustible—and are something even more. He explained that while it is “[the] virtual [which] possesses the reality of a task to be performed or a problem to be solved,” the problem itself “orientates, conditions and engenders solutions” (1994, p. 212). Problems are filtering manifolds through which the virtual *becomes* as it flows. The ’software crisis’ became a term used to describe the shocking expense and poor results produced during the initial attempts to use computing to solve problems. While part of the crisis was, as Cambell-Kelly et al. (2014) explain, “finding enough experienced programmers to do the work” (p. 174), the other was principally the speed at which the computing hardware changed, and then the reliability of what was produced at all.

www.manaraa.com

While the divide was initiated in large part by Atchison et al. (1968)'s exclusion of explicit engineering education as a strategic move to reduce the amount of interference from outside entities in defining and legitimizing the study of computation, the effort was only partially successful if viewed as an expression self-interested and externally motivated technical regime. Around the time of the 'software crisis' during the 1960s, corporations like IBM or Remington Rand worked to exert influence over matters of education at universities by not only furnishing the computational resources and machines necessary for Computer Science's curriculum, but by introducing programming languages and practices that would define the way problems and solutions were discussed, defined, and solved. As the computing industry emerged during the post-war period of the 1950s and early 60s, issues of authority and power—to both define the ways in which computing was used, and to label and describe what constituted 'programming' and 'expertise' in a push to professionalize—emerged which pitted not only large US associations like the Data Processing Management Association (DPMA) and Association for Computing Machinery (ACM) against one another, but also included manufacturer's like IBM and Remington Rand and others in bids to determine how computers would be used, who would use them, and toward what purposes.

The so-called 'software crisis' was a product as much of the difficulties of accruing expertise inherent in the material encounters of human and machine through both hardware and software processes (like maintenance and development, respectively), and the fierce competition of industry powers for authority over the nascent field which used computers as a medium through which to think about and solve problems, it was also a product of relations of speed and time, to put a Spinozist (and Deleuzian) spin on the event. The problem was never just about 'programming,' but of the speed at which hardware evolved and software stagnated.

المنارة للاستشارات
www.manaraa.com

Programming, as Brooks (1995) would explain later, was a small component of a process comprised of many components, and the 'expertise' issue at the heart of the software crisis was fundamentally about working methods and standards in relation to constant movement. Figure 3.2, taken from the RAND Symposium "On Programming Languages," depicts programming's cost relation to other factors like training, testing, maintenance, documentation, and so on:

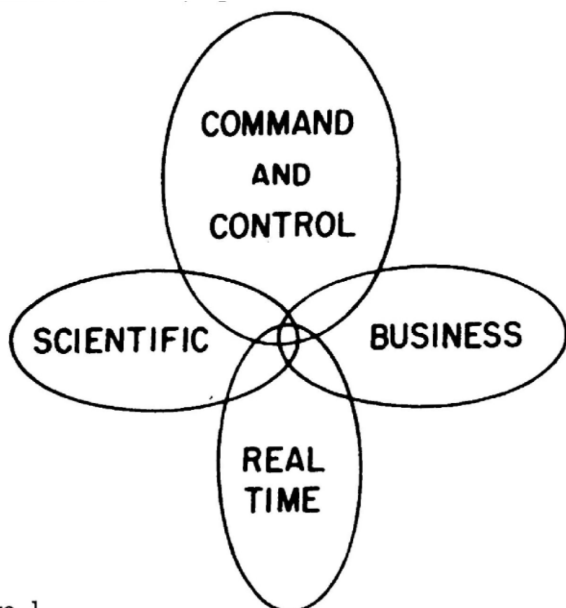

Figure 3.2: The formula for cost analysis forwarded at the 1962 RAND Symposium, "On Programming Languages" (Patrick et al., 1962, p. 25).

From the perspective of a technical regime, the factors alluded to in Figure 3.1 during the "On Programming Languages" symposium show a small area of intersection between business and scientific computing.

The RAND Symposium occurred in response to a rumor that the Department of Defense wanted to "standardize on a language for Command and Control" (Patrick et al., 1962, p. 25). In 1962, COBOL was 3 years old, and language standardization—the idea of cross-platform compilation, or in modern parlance, 'write once, compile everywhere'—was desired to reduce the cost of computation. Beyond industry or governmental inertia, the idea that a language

المنارة للاستشارات
www.manaraa.com

becomes standardized when it is used by a large enough player to the exclusion of others, the need for standardization was summarized neatly in a statement made by Patrick that spoke to a line of intersection between many plateaus that would influence and shape the software engineering's plane of reference and technical regime six years later:

> The problem gets refreshed every day every time you order a computer. Thirty-six months from now you probably won't have anything installed that you have installed now. In other words [sic] you'll get a fresh start in 36 months. (p. 28)

The pace at which computer hardware evolved during the 1950s and 1960s moved swiftly, but the *operating methods* and values within the nascent and competing technical regimes of science and management had not yet standardized: essentially, the constantly 'new' tools—assemblers and compilers—and operating systems and computational architectures were an impediment, rather than facilitator, of positive development and change. Yet, while the RAND Symposium members recognized the desirability for a 'standard language,' i.e., one whose syntax and outcomes were "maintained and enforced" by means of "some sort of committee agreement," they argued about what the threshold for standardization was, and upon whose authority calls for standardization relied. With standardization, we see a move toward diagrammatization that Deleuze and Guattari discussed (Watson, 2008), as well as a move toward fixing or inscribing axioms upon a disciplinary plane of reference. Further, the software crisis demonstrates one of the principles argued by Deleuze and Guattari (1994) regarding the speeds at which disciplines work: philosophers, as they survey a plane of immanence upon which concepts reside, survey at infinite speeds; science works by slowing a phenomenon down such that it can be processed, ordered, and reproduced. The respective planes for both disciplines are ways of "confronting chaos" by "laying out a plane, throwing a plane over chaos" (Deleuze & Guattari, 1994, p. 197).

المنارة للاستشارات
www.manaraa.com

That the rigorous mapping of axioms across a plane of reference was compromised by the rapidity at which computing changed is no surprise, in hindsight. The pace of hardware development and software integration of computational resources into academic and corporate states of affairs left little time for the axiomatization of propositional functions, making it easier to see how programmers *could only be* idiosyncratic, self-taught employees: no formal basis of praxis could initially be established, and so chaos dominated to the extent that agonism between theory and praxis existed, such that the failure of a stable plane of reference formed the basis of the software crisis.

From managerial praxis, computation's nascent and unstable plane of reference was inherently navigated and learned 'on the job.' Programmers had to learn how to program the specific machines their employers owned; they also had to learn their employer's contexts, use-patterns, and user requirements for the systems they were asked to automate, improve, and deploy. At one point during the RAND Symposium (Patrick et al., 1962), a participant indicated that there had been as many as 90 "common languages" used across the many different computer mainframes bought and deployed by customers across the United States (p. 26); programmers were expected to learn and become proficient in a *relative* style of engineering—an issue that is not unlike the circumstances they find themselves in today. Idiosyncratic expertise was problematic from a managerial perspective due to not only to its ability to upset org charts and structures of authority and power, but for its expense and inherent unpredictability (if not unreliability). But 'expertise' is, crucially, where consideration from a Deleuze and Guattarian perspective produces insights: the 'software crisis' was the product of an unstable plane of reference that undermined the ability for early programmers, managers, and scientists to think and operate clearly. For science, a plane of reference filters and delimits the pure virtual (e.g.,

المنارة للاستشارات
www.manaraa.com

chaos) comprising matter and its immanent and emergent possibilities; if a plane of reference is in flux, it stands to reason that the virtual, in all of its possible forms, influences the actual. Deleuze and Guattari (1994) explain that "[knowledge] is neither a form nor a force but a *function*: 'I function'" (p. 215) and that ultimately, knowledge, from a scientific perspective, is for

> setting limits that mark a renunciation of infinite speeds and lay out a plane of reference; assigning variables that are organized in series tending toward these limits, coordinating the independent variables in such a way as to establish between them or their limits necessary relations on which distinct functions depend, the plane of reference being a coordination in actuality; *determining mixtures or states of affairs that are related to the coordinates and to which functions refer*. (emphasis added)

Without a stable plane of reference, limits are unknown; variables cannot be fully identified, preventing coordination of variables and other dependent states; the mixture of the 'function' of knowing, the ability to operate from a plane of reference in the real, would always be compromised. Hence, as Schönher (2013) explained, "[laying] out a plane is the condition for that plane to rise to the power of thinking and creating" due to its ability to intercede in the actual *and* give shape to the virtual, to delimit relations in states of affairs (p. 32). A plane of reference shapes the functioning of thinking and knowing. It follows that an unstable plane of reference produces turmoil, which is evidenced succinctly by Daniel D. McCracken's (1961) recognition that not only was "[the] main trend in the human side of computing … the explosion in the number of humans" (p. 9), but the ability for the industry to both maintain and induct existing and new experts was failing. Programming, as noted above by the RAND Symposium and Brooks, was one part of the process of software development; to solve problems reliably,

المنارة للاستشارات
www.manaraa.com

programmers needed to learn how to design systems, and "systems work is not so much a body of factual knowledge, as an approach to problem solving—and no one knows how to teach the problem solving approach" (McCracken, 1961, pp. 9-10). While it can be systematized, 'problem solving,' computationally or otherwise, is fundamentally rhizomatic, intersectional, and experiential.

This section described the issues managers faced during their initial encounters with computation and the programmers required to 'solve problems computationally' within corporate environments. De-fetishizing source code and programming involves recognizing the communicative and technical practices involved in acquiring expertise, which is and has been a historically human feature of production. Expertise afforded software developers a disruptive form of autonomy within conservative business cultures, and even 'expert' software developers of the day were still considered to produce unreliable, buggy software, ergo the software crisis. The issues of manageability of programmers, and the confounding of their initial conception as translators and passive receivers, could only be mitigated through systemic reductions of expertise. The next section describes how 'software engineering' was the academic and industry response to the theory and practice duality in the expertise domain of programmers of the software crisis.

Emergence of Software Engineering

> In many creative activities the medium of execution is intractable. Lumber splits; paints smear; electrical circuits ring. These physical limitations of the medium constrain the ideas that may be expressed, and they also create unexpected difficulties in the implementation. … Computer programming, however, creates with an exceedingly

المنارة للاستشارات
www.manaraa.com

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

This section explains how software engineering became formalized in response to industry and academic factors. It establishes how the identity of 'software engineer' or 'software developer' came to encompass what it does, while also entangling many practices and processes that, once integrated into media studies conceptualizations of 'software,' can only de-fetishize 'programming.' The emergence of 'software engineering' in 1968 was as much an attempt to increase the reliability of 'software production' by applying engineering practices (and thinking) to its development (thereby reducing its costs and increasing the predictability of its outcomes), as it was to reduce the authority and power of its practitioners from the impositions of unassailable expertise. For corporations navigating the software crisis, the costs and expertise required to produce software needed to be industrialized, such that any programmer could be substitute for or with any other programmer, so that timelines could be forecast and costs predicted; but for academics, as evidenced by Atchison et al.'s (1968) Curriculum '68, the reliable production of software *products* was not the direct pedagogical concern of Computer Science. Rather, CS would inculcate a form of computational thinking which was focused on abstract rather than practical considerations, and explicitly set practices like budget estimates, needs assessments, requirement gathering, planning, testing, and documentation skills aside. And while modern descriptions of computational thinking incorporate words like 'engineering' (Wing, 2006) into their explanations, they tend to exclude the realities of the praxis encapsulated

المنارة للاستشارات
www.manaraa.com

by the term by focusing on the benefits of the abstract, theoretical concerns of disciplinary Computer Science. So, the fact that software engineering and Computer Science were both formalized during the same year is significant, rhizomatically, because it shows how lines connection were trimmed, and lines of flight continue to persist and resist modern summations of skills and categorical labeling: in many ways, Computer Science is the study of computation to the exclusion of the software corporations expected it to teach how to produce. This means that, just as programmers had to learn how to program in the scope and context of their specific corporate environment, programmers are still—despite some courses at contemporary universities in 'Software Engineering'—expected to learn the essence of their praxis on the job. A status quo was established at some point in CS' past that continues into contemporary times. The tension between corporate and academic interests leading to the development of 'software engineering' was therefore one of accountability and resistance: 'software engineering,' as it was called at the eponymous NATO summit in 1968, was an attempt to formally map out a plane of reference comprised of axioms from managerial and industrial praxis and Computer Science to make it both manageable and teachable.

But what makes 'software engineering' an interesting praxis to study from a Deleuze and Guattarian perspective is the nature of software itself, with its conjunction with axiomatic and political impositions, its relations and definability as rhizomes and assemblages, and its ontological and epistemological consequences. Mahoney (2008) explained, in *What Makes the History of Software Hard*, that the "history of software is the history of how various communities of practitioners have put their portion of the world into the computer" (p. 8). What makes the history of software hard, according to Mahoney, is that software is perpetually the product of a 'legacy,' of the historical patterns and trends which bring it into being. From a Deleuze and

المنارة للاستشارات
www.manaraa.com

Guattarian (1987) perspective, software might be, if we recognize it as being both a product of its legacy and its immediate expression, an always-processual *adjacement*, which is the dynamic form of an assemblage: it is always coming into being. So, software, as it is executed on a computer, is always brought into being through its compilation into machine instructions or bytecode, and its legacy is always evidenced in the trends and patterns and libraries referenced in its source code by the axiomatized thought-form of its language. Software might be the "nomadism of those who only assemble" (p. 24), because it is never only its source code, its machine instructions, or its execution ontologically; software is both a historical precedent and antecedent and is thus an expression of an episteme, a telos through an axiomatized plane of reference that brought it into being while simultaneously allowing it to serve as a basis for the development of the next becoming, the next "version" of an application, the next exhausted iteration of a problem's inexhaustibility.

If software has such a fluid definition—as static source code or compiled machine instructions, as a dynamic product, a process, a telos, as a legacy—software engineering is necessarily an imposition or filtering of chaos, the stretching of a sieve-like plane of reference over the pure virtual "thought-stuff" comprising 'software' alluded to by Brooks (1995). The axiomatization of software production, of programming, began nearly as soon as stored program computers like EDVAC and Cambridge university's EDSAC were conceived and implemented in a generally programmable way. "Baby," the Manchester Small-Scale Experimental Machine (SSEM) went live in June of 1948 and is credited as the world's first stored-program computer, the installation and delivery of EDSAC and EDVAC in 1949, to Cambridge and the U.S. Army's Ballistics Research Laboratories, respectively, provided a basis for automation that would require increasingly sophisticated tools and conceptual models. In 1948 at Cambridge, David

المنارة للاستشارات
www.manaraa.com

Wheeler, a PhD student, was tasked with solving the "programming problem," and began working on the first symbolic converter, a program of 30 instructions which converted logical (and human readable) statements into binary machine instructions (Cambell-Kelly et al., 2014, p. 169). Wheeler's program, called "Initial Orders," solved the problem of readability (after a fashion) by allowing other programs to be written using tools like assemblers, meaning the task of writing in 'machine instructions' or pure binary could be left to the computer. However, Wheeler's symbolic converter exposed another problem, that of "getting programs to work correctly." Cambridge's solution would cement, as Mahoney (2008) already noted, the 'legacy' inherent in software: as "[it] was realized that many operations were common to different programs," e.g., programmers would often be asked to determine the square root of a value, or to print a specific line of text on a display, the Cambridge group developed a library of subroutines (Cambell-Kelly et al., 2014, p. 169). Programmers would develop programs that consisted of a smaller amount of original code by relying on larger amounts of (ideally) tested and proven subroutines; this development, "[the] idea of reusing existing code was and remains the single most important way of improving programmer productivity and program reliability."

Significantly, the model established at Cambridge has not changed. Using subroutines and libraries of code are a primary form of mediation in programming; it is arguably correct to state that programmers are mediated as much by their tools and libraries as they are through the programming languages they use. As a trope, many beginning programming textbooks have students compile a program which prints a statement, like "Hello, world!" to a display. Deitel & Deitel's (1994) textbook on the "C" programming language lists the program in Table 3.2 as the first program a student might try compiling and running:

المنارة للاستشارات

www.manaraa.com

Table 3.2: A slightly modified first C program from Deitel & Deitel's (1994) textbook, adjusted to include the stdio.h header so that standard input and output subroutines, like "printf," are available to the program, making it buildable (p. 24).

```
#include <stdio.h>

/* A first program in C */

int main()

{

        printf("Welcome to C!\n");

}
```

Even the simplest programs require libraries to function. To get Deitel & Deitel's program to compile and run in Microsoft Visual Studio 2017, I had to modify it slightly by incorporating an 'int' (4-byte, 32-bit signed integer capable of representing positive and negative whole numbers) as the return value of main() because Microsoft's C++ compiler required the main procedure and entry point of the application to have a return type; additionally, I had to include "stdio.h," which is a header file to C's common input and output methods (like "printf," print function). Without including stdio.h, the program in Table 3.2 would have no way print text to a standard console display without the programmer deliberately writing a new implementation using low-level methods through inline (or linked) assembler statements, which might have worked with MS-DOS (Microsoft Disk Operating System) at the time Kittler (2013) was writing "Protected Mode" in Word Perfect, but would be prohibited by modern Operating Systems (OS), which limit a program's access to underlying hardware for security and virtualization reasons (among others)—somewhat unironically the reason 'protected mode' was implemented by Intel, to the chagrin of Kittler in his eponymous article. Programmers program *through* libraries and think

المنارة للاستشارات
www.manaraa.com

through their tools, which are integrated and organized in memory—not to mention linked by to the program's executable file—using principles designed by Wheeler in the early 1950s. Cambridge also published the first programming book, *The Preparartion of Programs for an Electronic Digital Computer* in 1951, which "set the programming style for the early 1950s, and even today the organization of subroutines in virtually all computers follows this model" (Cambell-Kelly et al., 2014, p. 170). The role of 'legacy' in the development of software and the emergence of software engineering, of the "many histories" and communities of practice Mahoney (2008, p. 8) noted, shows lines stretching across a plateau rhizomatically, like rings of a tree, or layers of an onion's cross-section.

The NATO Software Engineering Conference in 1968 was an attempt to control some of the growth of the software development plateau. While phrases like 'standards and practices' were written about extensively prior to and during the software crisis in journals like Datamation and Communications of the ACM, the reality of those things were relative to specific local contexts, like *a* business or *an* academic computing center. By conceiving of the software crisis as a plateau to help us understand its development, progression, and on-going consequences, the power of Deleuze and Guattari's approach to history is evidenced by a tracing of the intersections of relative attempts at standardization. Essentially, and with some obviousness, the legacy of one practice can influence and shape other intersecting practices, but the attempts to standardize the direction of the event of software development can be thought of as culling points of opportunity, or lines of flight, upon its plateau. With a sufficiently axiomatized plane of reference such culling deliberately favors the selection of an axiom and ordinate and seeks to reduce the likelihood of alternative selections. But what happens when 'first principles' are themselves in flux, and the plane of reference is unstable? Rhizomatic growth occurs in a form

المنارة للاستشارات
www.manaraa.com

where many concentric rings, representing communities of practice, compete at points of intersection for dominance: what works best; what can be shown to work best; what potential methods can lead to something working best?[2] Imposing control over the ordinates used to reference axioms controls ontological outcomes, and those controlling the selection of ordinates shape its epistemology. The NATO conference participants recognized that they were addressing a polemic issue: '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 & Randell, 1968, p. 8). The term was provocative because, despite being "fully accepted that the term software engineering expressed a need rather than a reality" (Randell, 1996), the material reality of computation—its in-betweenness, its virtuality, its nature as 'thought-stuff'—made it difficult, if not impossible, to test in the ways demanded by the epistemologies bound to those implied by the 'engineering' label.

The controversy was addressed by Edsger W. Dijkstra some years later in a way that shows how the event of the 1968 NATO conference is ongoing—how the controversy of the label continues to resonate and intersect with neighboring plateaus. As Dijkstra (1988) noted in a speech delivered in Austin at the University of Texas, aptly titled "On the cruelty of really teaching computing science," that computers were indivisibly complicated *and* capable of expressing immensely dense datasets qualified them as "radical novelties" that could only sufficiently be appreciated and taught using an "orthogonal method" of pedagogy: "[coming] to grips with a radical novelty amounts to creating and learning a new foreign language that can *not*

[2] What is interesting here is that the plateau and the plane of reference of software development ensnares and integrates both *asemiotic* and *semiotic* significations: "One has, with initially a kind of split personality, to come to grips with a radical novelty as a dissociated topic in its own right. Coming to grips with a radical novelty amounts to creating and learning a new foreign language that can *not* be translated into one's mother tongue" (Dijkstra, 1988).

المنارة للاستشارات
www.manaraa.com

be translated into one's mother tongue." Dijkstra was a pioneer in Computer Science, abandoning a PhD in physics in favor of the nascent discipline in 1959. To him, programming during the era of the software crisis was troubled principally by the idiosyncratic expertise inculcated by its practitioners: "[programmers] too often saw their work as temporary solutions to local problems, rather than as an opportunity to develop a more permanent body of knowledge and technique" (Ensmenger, 2010, p. 112), and so his later statements worked to emphasize the intellectual opportunities programming yielded. However, that a programmer worked with 'radical novelty,' or expressed it, meant that, as a *job requirement*, they had to possess the ability to work with numbers and datasets that

> totally [baffle] our imagination, [and have] to be bridged by a single technology. [The programmer has] to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before. Compared to that number of semantic levels, the average mathematical theory is almost flat. (Dijkstra, 1988)

The point Dijkstra was making is that computation is essentially indivisible from the perspective of managed complexity. Complexity is typically managed through division: large things are divided into smaller things; if the smaller thing is still too complex, it is further divided. Yet, from Dijkstra's perspective, the abstract and theoretical basis of Computer Science cannot be simplified: its 'semantic levels' are too deep. So, of software engineering itself and the effort to teach principles and techniques to aid development, the attempt to systematize or 'engineer' software is merely an attempt to hide computation's complexity:

> A number of these phenomena have been bundled under the name "Software Engineering". As economics is known as "The Miserable Science", software engineering should be known as "The Doomed Discipline", doomed because it cannot even approach

www.manaraa.com

> its goal since its goal is self-contradictory. Software engineering, of course, presents itself as another worthy cause, but that is eyewash: if you carefully read its literature and analyse [sic] what its devotees actually do, you will discover that software engineering has accepted as its charter "How to program if you cannot." (Dijkstra, 1988)

Essentially, the argument Dijkstra makes is that the study of computation, the science of information and algorithms, is too complex to be divided and further subdivided into forms allowing people to work with parts of the discipline; rather, its practitioners must work with wholes. His crux is that one should use theory to find the true nature of the problem such that its solution can be expressed for all problems of a type, i.e., in speaking about the placement of dominos on a checker-board pattern of arbitrary size, the answer is to "deal with all elements of a set by ignoring them and working with the set's definition." This is interesting, because it, on one hand, implies an orientation of the 'computer scientist' toward their disciplinary plane of reference, e.g., all categories of knowledge and their referent axioms must avail themselves always, mediated through a disciplinary practitioner, to the study and consideration of a problem. And on the other, that there is a kind of transcendent, rather than immanent truth expressed through mathematical reasoning, which the 'science' of computation is ultimately a form of.

The rhetoric of the software crisis was not empty. While Curriculum '68 cemented the theoretical basis for Computer Science, the NATO Software Engineering Conference in 1968 exposed a gap, which is implied in Dijkstra's statements about the complexity of Computer Science. The problems associated with software design were fundamentally rooted in scalability, and what happened—and continues to happen—when the scope of a solution grows exponentially in response to the problem (or manifold problems) it attempts to solve. Considering that programmers at the time of the NATO Software Engineering Conference might

المنارة للاستشارات
www.manaraa.com

have had a degree in a related field, like mathematics, physics, or some form of engineering or accounting, they came to corporate jobs deficient in the skills they would need to learn to be proficient in those environments. As Brooks (1995) came to realize while managing the development of IBM's OS/360 operating system during the mid-1960s, programmer labor was not directly proportional to a task. "The bearing of a child," he famously quipped, "takes nine months, no matter how many women are assigned" (p. 17). More programmers did not translate into faster development, or even better reliability, and recognizing if an application 'solved' anything required new ontologies and epistemologies to interpret development outcomes. The production of software dips into and out of virtual and actual domains, and at some point, the reasonable response to manage the complexity of a problem is division. Individuals at the conference, like Mr. A. G. Fraser, recognized that

> One of the problems that is central to the software development process is to identify the nature of progress and to find some way of measuring it. Only one thing seems to be clear just now. It is that program construction is not always a simple progression in which each act of assembly represents a distinct forward step and that the final product can be described simply as the sum of many sub-assemblies. (Naur & Randell, 1968, p. 10)

At the height of the software crisis, programmer labor not only confounded traditional metrics and practices, but programmer output was more than the sum of its parts. Dijkstra's argument about complexity—the indivisible whole—is reasonably true if a program is considered as an immanent expression of relations, in a Deleuze and Guattarian sense, or how Hayles (2005) or DeLanda (1997) might explain it. In mechanical engineering, repeated physical measurements of material strengths and physical interactions produce a model which frames expectations, such that a design's success can be confidently predicted; for software efforts, it was difficult to

المنارة للاستشارات
www.manaraa.com

anticipate the outcome of the assemblage of parts many programmers produced. At the conference, Dr. E. E. David recognized that "there are good reasons why software tasks that include novel concepts involve not only uncalculated but also uncalculable risks" (Naur & Randell, 1968, p. 9), leaving corporations to find a way to efficaciously deploy computational resources, and discovered that the human costs associated with the purchase of those resources far outpaced initial hardware expenditures[3].

So, while Dijkstra came to view software engineering with suspicion (if not outright derision), Cambell-Kelly et al. (2014) noted that "the theoretical concerns of academic computer scientists were not always seen as being relevant to the problems faced by working corporate programmers" from a business perspective (p. 184). The degree to which managerial and Computer Science's referential planes intersected determined not only respective truth values—what truth should look like, whose truth it should be, types of outcomes, pedagogies, etc.—but the reality of expressing those truths, and of moving from virtual or diagrammatic domains into actual and concretized domains. The production of software in 'the real-world,' for lack of a better term, required a way to manage and systematize the design and implementation of a real-world (or emergent) problem using abstract principles, and had to axiomatize a broad set of asignifying *and* signifying functions and ordinates because their propositions had to encapsulate not only the abstract, diagrammatic principles and processes of Computer Science, but the ideally diagrammatic but perpetually exposed and compromised systems of signification which comprised practices like 'needs-assessments,' 'requirement-gathering,' 'labor management,' and the authority and power of expertise. At some point, a 'real-world' design is compromised by signification, in the sense that it fails to capture what Dijkstra (1988) called "down-to-earth

[3] Famously detailed in Brook's (1995) account of IBM's development of OS/360.

المنارة للاستشارات
www.manaraa.com

mathematics" in his speech to the Computer Science department at UT Austin. Dijkstra attended the NATO conference for software engineering and recognized at the time that "the massive dissemination of error-loaded software is frightening" (qtd. in Naur & Randell, 1968, p. 9), but dismissed attempts codified by 'software engineering practices' twenty years later as false solutions. Yet, those solutions persist because they serve the needs and requirements of those needing to forecast and measure programming labor and practices and are themselves the emergent expressions of praxis from their own plane of reference, their own plateau.

Software engineering is neither purely managerial praxis and theory, or computational praxis and theory, but was born of both disciplines, yet encapsulates something more in its attempts to mitigate expertise. Computer scientists, managers, and programmers attended the NATO conference in Brussels, where representatives from "computer manufacturers, universities, software houses, [and] computer users" were invited to speak and work toward developing better understandings of software design, production, and service (Naur & Randell, 1968, p. 8). One of the major developments—for both good or bad, depending on whom one asks (c.f. Dijkstra, 1988)—was the development of 'programming methodologies.' Today terms like 'waterfall,' 'agile' (i.e., "extreme" programming), 'DevOps,' and 'continuous delivery' are used to describe the practices teams, departments, or companies use to conceive, design, implement, test, document, and deliver software. Software methodologies evolve as developers and managers continue to revisit the problem Fraser (Naur & Randell, 1968) noted, of adequately identifying and measuring "the nature of progress" toward completion, using new tools and communicative strategies to ideally increase the accuracy of when the "sum of many sub-assemblies" becomes the "final product" specified and designed at the outset of work. Such methodologies not only structure the programming tasks and shape the resulting code

المنارة للاستشارات
www.manaraa.com

programmers produce provides a mechanism that employs relative levels of expertise at different project scopes: junior developers are substitutable and are used to perform most of the programming, while experts architect and design the solutions. Mr. K. Kolence defined a software design methodology as being "composed of the knowledge and understanding of what a program is, and the set of methods, procedures, and techniques by which it is developed" (in Naur & Randell, 1968, p. 15). One of the first programming methodologies was 'structured design,' which "reflected the belief that the best way to manage complexity was to limit the software writer's field of view" (Cambell-Kelly et al., 2014, p. 185); the method was alluded to by B. Randell in a paper presented at the conference, *Towards a Methodology of Computing System Design*, which called for "a structuring of the design process," which emphasized a phrase (and concept) used by Dijkstra, Zurcher, and Randell in other publications, called "level of abstraction" (Naur & Randell, 1968, p. 118). A level of abstraction essentially allows problems, tasks, and implementations of lower levels of code to be considered 'solved,' so that operations on the current level can be written, and issues at higher levels can be ignored. As a level is finished, its "primitives … are provided by the processes of the immediately lower level," so that "each level … is … a set of solutions … to a set of problem areas which the designers have chosen to regard as being closely related." At a functional level, the diagram of a structured design will differ wildly from an unstructured one because each level of abstraction controls—in a properly designed and implemented system—access to the data and methods of their respective level: low-level, essential implementations can be handled by experts, while higher-level tasks can be programmed by entry-level developers because they use procedures and data that are ideally tested and proven to work. Layers control, by design and implementation,

المنارة للاستشارات
www.manaraa.com

what developers working on other layers can see and use: without abstraction, developers could effectively 'touch' (i.e., change) all aspects of a system.

Programmer manageability transitioned through the software crisis from a reputation of idiosyncratic and unassailable expertise, toward one of substitutability, due in large part to the discovery of the structured program theorem in 1966. Conceptually, while structured software design and structured programming involves a great deal more than 'levels of abstraction,' it is notable for being the first step towards limiting a programmer's 'vision' by containing a scope of work to only that which a programmer was required to implement, a fundamental tenet of managing complexity *and* the labor that goes with it (Linger et al., 1979):

> the rediscovery of software as a form of mathematics in a deep and literal sense is just beginning to penetrate university research and teaching, as well as industry and government practices. The forcing factor in this rediscovery has been the growth of software complexity, and the inability of informal software practices and management to cope with the complexity of today's challenges in software. (pp. vii-viii)

Structured programming not only attempted to classify programming as a form of mathematics but attempted to harness the work and ultimately the communicative practices programmers performed. The work of Dijkstra, not incidentally, was crucial to the rise of structured programming as a methodology (p. viii). What a programmer sees, or can see, defines the extent to which they work, and that 'vision' describes not only their level of expertise, but the extent to which they are trusted within the scope of an implementation to make changes *by management*. 'Vision' works its way through descriptions of levels of abstraction through words like 'implementation hiding' (which is a crucial element of interfaces, which are public contracts describing libraries of methods and data expose for programmers, while hiding their

المنارة للاستشارات
www.manaraa.com

implementations from observation). By not having to see everything, a developer should only see what they need to see to complete a given task. Provided the proper libraries for a given scope, one programmer should be able to do the work of any other programmer with a similar level of expertise. That entire programming languages, e.g. Pascal in 1971, were built around the concept of structured design and taught to undergraduate Computer Science courses for more than 20 years indicates the desirability of control (Cambell-Kelly et al., 2014, p. 185), not only over the problem to be solved, but the programmer working on the solution. Software design methodologies have always worked, from the outset of their conception, to realize the "ultimate goal" of a "'software factory' complete with interchangeable parts (or 'software components'), mechanized production, and a largely deskilled and routinized workforce."

The combination of 'software' and 'engineering' was (and is) controversial not only because it is *not* about the discovery of the true through mathematical reasoning or the holding up of transcendental ideals in a Kantian sense, but especially so in that it does not have the same kinds of empirical and therefore measurable outcomes that other engineering disciplines enjoy. Dijkstra's statements about complexity and his overall attitude toward software engineering illustrates the tension that persists between managerial and Computer Science and is evidence of distrust of the methods used to design and implement software. The tension between managerial and Computer Science—especially as it relates to the software crisis—is that ultimately the economic production of software is about the reliable implementation of a product that meets a set of needs while (hopefully) solving a problem that someone—usually a customer—(hopefully) has. Just as there are products that exist which have no purpose, there is poorly conceived software that, through misinterpretation, obstinance, or ignorance during the requirements gathering and assessment phases, progressed unwisely toward diagrammatization and

المنارة للاستشارات
www.manaraa.com

implementation. Atchison et al. (1968) listed three broad categories in Curriculum '68, "Information Structures and Processes," "Information Processing Systems," and "Methodologies" (pp. 154-155), and it is difficult to believe that any pedagogical system is capable of imparting an ever-expanding totality of knowledge to a student; software engineering, as a *human* and *social* praxis, was tasked with—at the outset of its conference—finding a way of bridging and translating aspects of Computer Science's plane of reference into ontologies and epistemologies that valued and defined 'truth' differently, *and* was designed to place impositions on its human actors to make them controllable, predictable, and replaceable, while also allowing those skills to be deployed across organizations encompassing multiple cultures and needs. The gap between the academic and corporate planes of reference, where computation is concerned, resides in the uncomfortable blending of signifying and asignifying functions and propositions conjoined with praxis that must produce both signifying and asignifying results, which will be explored in the next chapter.

This section described how software engineering came to be considering industry and academic practices and considerations. It established how the identity of 'software engineer' or 'software developer' came to be typified by programming, which is emblematic—essentially a stand-in—for the broad array of practices that academics and corporatists expect from software developers. This section has shown how 'software engineer,' regardless of its inherent polemics as an identity, is an intersectional and rhizomatic community of practice that has developed its own plane of reference in response to both the inceptive demands placed upon it from experiences throughout the software crisis of the 1950s and 60s and ongoing requirements for implementation efficacy and manageability.

المنارة للاستشارات
www.manaraa.com

Software Engineering as a Plateau

Understanding software engineering as a thought form distinct from Computer Science and managerial praxis is the first step toward de-fetishizing 'programming' and 'source code' in media studies scholarship. Many demands are made of software developers, and as a result, the axiomatized plane of reference has expanded and contracted rhizomatically across its ongoing plateau. At first programming around the time of EDSAC "was an issue that was very much taken for granted by the majority of early computer projects" because "the emphasis was on the hardware, so that it was only when a machine sprang into life that the business of programming was seriously considered at all" (Cambell-Kelly, 1992, pp. 18-19), and the expertise it engendered did gift a disruptive amount of authority to agents outside of traditional managerial structures. However, programming became relegated to one of many practices, in a way, after those managerial structures evolved to clip certain lines of flight that expertise afforded those software developers, effectively reducing their agency in corporate environments by collaterally reducing their expertise. Now, while modern software developers are asked to perform many duties due to the nature of the interaction of Computer Science and managerial praxis, they use software developers use many practices—like Software Development Life Cycles and Software Testing Life Cycles (SDLC and STLC, respectively)—to produce reliable software, issues that will be explored at length in the next chapter.

Programming, while important in the sense that it produces what a computer executes, i.e., software, is less important that adequately and appropriately understanding the problems the software should solve; de-fetishizing programming allows software and its developers to seen as agents of many practices—as points of intersection between competing and complimentary planes of reference—which are more important than choosing variable names or commenting

المنارة للاستشارات
www.manaraa.com

lines of code. The practices involved in SDLC methodologies, for example, involve translating problems that reside in signifying domains into solutions which are expressed in asignifying domains. Arguably, the root of good software development is not only a grasp of theory from the Computer Science side, but the ability to understand and communicate problems through managerial domains, by understanding people and their issues, and then turning those issues into products. Software and its developers are far more than 'programming,' and entire realms of communicative practices can be explored if the materiality of source code or programming as a form of writing is dismissed in favor of following a problem through whatever domains and communities of practice it intersects.

We can apply hermeneutics within problems to trace connections and discover emergent, immanent meanings as potentials within an assemblage of components, like industries, customers, limits of knowledge and material, and cultural requirements. At the outset of an interview Deleuze and Guattari conducted with Christian Descamps in 1980, "On *A Thousand Plateaus*," Deleuze asserted that concepts "should express an event rather than an essence" of an idea (1995, p. 25); events, he argued, can correspond to a specific date, a year, or a period of years, and "map out a range of circumstances" encapsulating "modes of individuation beyond those of things, persons, or subjects" (p. 26). Events are comprised of variables, which themselves represent lines along a plane, such that "everything has its geography, its cartography, its diagram" (p. 33). Diagrams—"[what] we call a 'map,' … is a set of various interacting lines"—reveal not only how an outcome came to be, as a historical effort and effect of a normalized, linear progression of interactions, but how the *shape* of the event depicted on the plane, its positive and negative spaces, the lines that connect as continuities or disconnect as orphaned segments, continue to affect and shape intersecting, and ongoing events. As a

المنارة للاستشارات
www.manaraa.com

sociotechnical regime, software engineering continues to interact with the events that lead to its inception as a whole; software development continues to be *difficult work* that is difficult to teach, difficult to manage, and unpredictable, and costly. Its planar orientation—the angle upon which it pivots between the plateaus of academic disciplinary theory and industrial managerial desires—has shaped its definition as a field such that, to become a software engineer, one must accede to being both secondary to theory and comfortable with being utterly reducible, a unit of predictable and replaceable labor.

For as tempting as it is to conflate software and software engineering with programmers and source code, it is arguably the communicative practices necessary to correctly gather and interpret requirements that determine the success or failure of the software designed and implemented for a particular purpose. The source code produced by programmers is only as good as how accurately the requirements were gathered and interpreted at the outset of a project, which is a form of expertise neglected in media studies' scholarship that attempts to define source code, or programming, or software. Programming is important, because at some point an application must be implemented, just as a part for a car is stamped and shaped from metal, but without the design, which clearly defines a solution to a problem, the source code or the car parts will fail. Successful software represents not only the culmination of distinctly technical expertise, but of communicative expertise as well, both within practices of design, and of labor management as well. Studying what goes into the production of effective software views great source code as a product of the mediating effects of the observations and communications necessary to endow one or more practices as a 'best practices.' This communication is a form of mediation that operates through a sociotechnical regime and involves at its core the transduction of signifying values into asignifying values, and vice versa.

المنارة للاستشارات
www.manaraa.com

Chapter 4: Transduction and Mixed Semiotics of Software Engineering

This chapter explores and defines transduction as the interfacing process of a sociotechnical regime. In it, I argue for an interpretation of Guattari's concept of mixed semiotics to show how a sociotechnical regime, like software engineering, relates to the world and incorporates the world back into itself. This chapter performs three tasks. First, it defines Guattari's concept of asignification and its role in the diagrammatism of software engineering processes, locating them in the plane of reference that organizes the limiting fields and assemblages of enunciation which define its sociotechnical regime. Second, it defines the concept of transduction as a process mediating the actual movement and conversion of values and their meanings into and out of asignifying and signifying domains. And lastly, it explains signification's role in transducive processes by specifically examining the difficulties inherent to the double articulation of a problematic. Transduction and mixed semiotics are a way to account for the problem of communicating problems, which is evident in the historic unreliability of software and the unpredictability of its implementation and delivery. Transduction allows media theorists to interpret the interactions of a sociotechnical regime like software engineering with external connections, like clients or users, integrating the nuanced ways an assemblage of processes brings software to life. Incorporating the many social and machinic connections (Chun, 2008) that realize code focuses attention on the ways, difficulties, and consequences of communicating problems between groups of software developers and customers, providing a way to study the mediating effects of problematics on collaborative tasks. Doing so de-fetishizes fixations on the programmer and 'source code' by exposing the communicative practices involved in realizing software.

المنارة للاستشارات

www.manaraa.com

Evidence for transduction and the interactions of signifiers and asignifiers is found in Guattari's (1984) repeated use of the term 'machinic.' Outlined in his "Machine and Structure" essay, for example, machinic invention becomes a motif which appears repeatedly throughout his work. For Guattari, technology was not only a thing but the theory behind the thing:

> The history of technology is dated by the existence at each stage of a particular type of machine; the history of the sciences is now reaching a point, in all its branches, where every scientific theory can be taken as a machine rather than a structure, which relates it to the order of ideology. Every machine is the negation, the destroyer by incorporation (almost to the point of excretion), of the machine it replaces. And it is potentially in a similar relationship to the machine that will take its place. (Guattari, "Machine and Structure," 1984, p. 112)

The 'machinic' implies processes of connection; the metaphor of the 'machine,' e.g., of desiring machines, of machinic invention, or machinic phylums are found in Guattari's contributions to his collaborations with Deleuze. The second sentence of the quote above indicates a recognition of the iteration technological developments undergo; the third, to the implementation of the new version. A type of development and domain of meaning exists that resists matters of purely human language, signs, semiotics, and linguistics: Guattari argued that, whether one's approach to such things were structuralist or enunciative, "[everything] happens as if the socius were thought to be folded within language" (2011, p. 25). Rather, he argued for a type of mixed semiotics that connected machines in ways that escaped signification: the human was not, and should not be, from Guattari's perspective, the center of history (Genosko, 2014). So, the conjunction of the machinic with semiotics—the role of the signifier, of encodings, and of diagrammatic processes—becomes "machinic semiotics," which "is a theory of history"

المنارة للاستشارات
www.manaraa.com

(Watson, 2008, p. 135), which becomes a way to describe connections and ruptures between materialistic contexts. What is premised here is that Guattari's contributions for a mixed semiotics, the integration of asignification into a semiotic expression, provide a mechanism to address Brown's (in Mitchell & Hanson, 2010) desire for an ideal materialism, which would be capable of showing "multiple orders of materiality … between a phenomenological account" of a user and a technology, "an archaeological account of the physical infrastructure of the medium, and a sociological account of the cultural and economic forces that continue to shape both the technology itself and our interactions with it" (pp. 59-60). Using mixed semiotics, generally, and paying close attention to the connections, overflows, and *distinctions* evident in diagrammatism is a way to interpret and understand the complex expressions produced by software engineers as an ideal materialism.

If sociotechnical regimes are "a particular type of machine," a way to encapsulate history, a way to demarcate a plane of reference from another (Guattari, 1984, p. 112), transduction describes the way its agents operate across boundaries, connecting to other regimes to affect things and events. Transduction accounts for the movement of signifying and asignifying content in ways describing the emergence of signifying meaning, of the play and evolution of language to determine our situations (Kittler, 1999) by the processes used to actualize them, which structures the media underlying future materialist exchanges. This is particularly evident for software engineering: software is at the root of the "metamedium" of computation (Manovich, 2013, p. 335). It stands to reason that an efficacious point of analysis for a sociotechnical regime are at its points of connection, between aspects of its incorporated practices and those of other agents and regimes, to determine the extent to which those connections are understood. By leveraging concepts from Deleuze and Guattari in both their individual and collaborative work, I seek to

المنارة للاستشارات
www.manaraa.com

provide theoretical insights into how connections might be analyzed by accounting for the factors increasing the likelihood for their miscommunication.

Asignification

This section develops a nuanced understanding of asignification, which is a part of Guattari's (1984, 2011) model of mixed semiotics. Understanding asignification in the scope of software engineering is important because of its fundamental role in mediating signifying interactions. As such, asignification is remarkable because it is inherent to systems of writing and book binding just as much as it is to typing on type-writers or programming source code for software, and doubly so for its ability to store a world of signifying meanings. From Guattari's perspective, while enunciations are often comprised of mixtures of semiotic forms, this section argues that systems organized around planes of reference (Deleuze and Guattari, 1994) are fundamentally (or try to be) diagrammatic and are therefore asignifying in nature. To establish this point about asignification and its significance to software engineering, this section begins by defining asignification; it then explores how software engineering communicates or makes by defining the role of diagrammatism in conjunction with Guattarian (2011) assemblages of enunciation; it then highlights the role of diagrammatic processes and Guattarian (2011) fields of consistency at work in McLuhan's (Year) assertions about automation and Kittler's (Years) discussions about 'digitalism' to highlight the role it broadly plays in software engineering. This section sets up the following section on transduction by forwarding the difficulties inherent to translating meaning from mixtures of semiotics that favor asignification into those that favor signification and back.

للاستشارات المنارة
www.manaraa.com

Asignification Defined

Guattari developed the concept of asignification to describe how nonhuman entities and scientific regimes communicated as part of his concept of mixed semiotics. His model for mixed semiotics comprises three areas, or types of singular enunciations: natural encodings; asignifying semiotics; and signifying semiotics. "Natural encoding[s]" are the "'a-semiotic transmissions of messages and codes" which occur "at the biological, chemical, and physical levels, and do not involve any kind of human language"; diagrammatics comprise "'a-signifying' semiotics" which are used by "information technology, science, and the arts [to transmit] ideas, functions, [and] intensities with *no need to signify any meaning*" (emphasis added, Watson, 2008, p. 47). Diagrammatic processes are of particular interest to sociotechnical regimes because they operate in 'machinic,' connective terms, comprised of asignifying expressions. Guattari's mixed semiotics allowed him to present modes of communication that resisted the "political motive … behind attempts to explain all encoding and message transmission in terms of linguistic language" (Watson, 2008, p. 47). This allowed scientific regimes to describe issues of materialism despite of signifying semiologies and provides a mechanism to resist processes of deterritorialization and overcoding.

Other interpretations of asignification exist. The editors of Footprint's 2014 issue dedicated to asignifying semiotics, "Asignifying Semiotics: Or How to Paint Pink on Pink," explained that "[asignifying] signs do not represent or refer to an already constituted dominant reality," but instead "simulate and pre-produce a reality that is not yet there" (Hauptmann & Radman). Examples of asignification and diagrammatism exist in Claude Shannon's original theory of information, which excluded semantic content from communication in favor of "a technical one based on uncertainty" or noise and entropy ratios (Genosko, 2014, p. 13). Even as

المنارة للاستشارات
www.manaraa.com

Shannon strove to define a model for the technical transmission of information which "[eschewed] meaning," Weaver imposed semantic content into Shannon's technical process "by interpolating a semantic received between the engineering receiver and the destination." Information transmitted between senders and receivers not only contend with entropy and noise at the level of the technical signal, but must, according to Weaver, through a process of "semantic decoding" incorporate noise at the level of 'meaning.' Guattari argued that the "dominant position" of information theory in linguistics, alluding to the Shannon Weaver model, focused on a "definition of language as merely a means of transmitting messages," and the adoption of the model was an attempt for linguists in the humanities to be "scientific," while also resisting the "interpenetration" of language with a "social field" (2011, p. 23). Genosko notes that Guattari "regarded information theory's 'skirmish' with meaning as a 'rearguard semiological conflict,'" implying that systems of signification are systems of domination (2014, p. 13). The transmission of information, in the Shannon & Weaver model, is a double-articulation in the sense that any transmission is a signal at the asignifying level—the presence or absence of electrons or photons for example—and the derivation of meaning at the semantic level; both articulations are subject to noise and entropy. The history of the software crisis of the 1950s and 60s demonstrated how the interference of one in the other—and vice versa—leads frustratingly just as easily to prosaic results as to paradoxical outcomes.

Asignification and diagrammatism are important for sociotechnical regimes because they provide a means to communicate in a realm that can, at times, mitigate semantic noise. Connections between humans are principally—but not exclusively—governed by signifying processes. From a Guattarian (1984, 2011) perspective, connections are formed between things in both concrete and abstract ways. Natural encodings have direct connections to the real world

المنارة للاستشارات
www.manaraa.com

as physical connections, such as nucleotide pairs in strands of DNA; asignifying connections made as part of diagrammatic processes connect to the real through sign-points, e.g., the notes of sheet music refer to an axiomatic of intervals of frequencies of sound corresponding to octaves of notes and their materialization as a key on an instrument, a note on a sheet of music, an expression in a state of affairs. For software engineering and other sociotechnical regimes, development efforts employ a mixture of asignifying, diagrammatic processes, organized as axioms, which are indices providing access to the functions and propositions of a plane of reference, which is a complicated way of explaining how the communication in and around a problem and its conditions relies on asignification to facilitate clarity, while still referring to material evident in a state of affairs.

Source code is an important type of communication for software engineering. Watson describes "computer code" as an example of a diagrammatic process, which begets the question: is it appropriate to 'read' code through semiotic practices from a Guattarian perspective? If we consider this issue from Guattari's perspective, look at the role of his work in his collaborations with Deleuze, and then examine software engineering as a plane of reference, we shall see that the implications of their work establish ontologies and epistemologies for technics that excludes, both by definition and through active resistance, processes of signification. While software 'is what it is,' at the level of asignification, it is enfolded into signifying processes when it interacts with the human developer, tester, technician, user. If code expresses the presence or absence of signals that have 'meaning' only insofar as those signals refer to a plane of reference encompassing the axiomatics they refer to (e.g., discrete logic, machine instructions, realization as electrical signals, electron flows through transistors), there is no 'signifying meaning' for it beyond its *use* as it comes into contact with 'the human' in its immediate state of affairs (e.g., a

المنارة للاستشارات

www.manaraa.com

computer, how the software is used, how it affects the subjectivation of its users). The mixture of processes of signification in software engineering include signifying connections made between humans and machines, communication between developers, the management of the human resources marshalled for the development of software, of the wide and narrow scopes of practices for responding to proposals, and generating needs assessments, writing user stories, can only affect future iterations of diagrammatization if those iterations are incorporated—axiomatized—into the plane of reference by which 'code' indexes—at which point semantics fall away. The diagrammatic process governing software development is the constant struggle between significations needing to overcode asignifications, while managing the noise that produces. Semantics can convolute and complicate the emergence of any product software developers work to actualize in a state of affairs.

Collective Assemblage of Enunciation

Guattari (1984) had a cynical perspective of technology, of mechanization, and particularly of computation, and is notable for being persuasively similar to a type of post-human rhetoric of 'recursivity' found in Kittlerian media studies. In Guattarian throught, 'the machine apparatus'—whatever it may be in a given context—exists independently of the human and can control and possibly subjectivate and reorder the human's position in a hierarchy to its eventual exclusion from that context[4]. The key to understanding Guattarian asignification lies in diagrammatism and diagrammatic processes in general, which reflect a kind of iterative network

[4] c.f. Kittler; the opening of *Media After Kittler* (eds. Ikoniadou & Wilson), which begins, "[the] question of technology has largely been conceived in humanist terms" (Ikoniadou, 2015, p. 1); the conclusion that Kittler's interpretation of 'protected mode' in the eponymous essay is indeed the reality that we are all somehow "subjects of Microsoft Corporation" (Sale & Salisbury, 2015, p. xxv). Kittler's explanation of 'protected mode' is an example of how Kittlerian-inspired research propagates a misinterpretation of a technological feature.

المنارة للاستشارات
www.manaraa.com

of decisions about what humans choose to axiomatize, and how they carried those decisions forward through machinic systems. Diagrammatism is a kind of "economy of signs" possessing "*sense without signification*" which effectively enables them to "simulate, 'duplicate,' and 'experience' the relational and structural nodes of material and social flows precisely at the points that would remain invisible to an anthropocentric vision" (Guattari, 2011, p. 59). While he believed "[operations] performed by workers, technicians, and scientists will be absorbed, incorporated into the workings of tomorrow's machine," and asserted that "[human] work today is merely a residual sub-whole of the work of the machine" (p. 113), which neatly aligns with some theoretical and rhetorical elements of the 'post-human' favored first by McLuhan (1994) and later Kittler (1999), the way diagrammatic processes are 'diagrammatized' and operate in mixed semiologies often produces narratives diminishing the importance of human selection and iteration on what becomes a function on a plane of reference, desirable for its efficacy, desired because it produces results that demonstrate value in accordance with the propositions functions must attempt to answer, e.g., engineering and scientific efforts. An example of "a non-signifying semiotic would be a mathematical sign machine not intended to produce significations; others would be a technico-semiotic complexus, which could be scientific, economic, musical or artistic, or perhaps an analytic revolutionary machine" (Guattari, 1984, p. 75). Taken as a technico-semiotic complexus which is "not *intended* to produce significations" (emphasis added), asignification, axiomatics, and the diagrammatic processes which rely on them require an expansion of their definition which includes the reciprocity at work within their formalizations and inscriptions upon a plane of reference. Software engineering practices are a reciprocating interplay of human interactions with axiomatic expression: one shapes the other, and vice versa. The reciprocity is evident in Guattari's nascent definition of asignification: an

www.manaraa.com

"analytic revolutionary machine," for instance, might today be evident in the practices and uses of Machine Learning (Alpaydin, 2016), which are axiomatized in the emerging sociotechnical regime of Data Science (Kelleher & Tierney, 2018). But what binds a 'data scientist' to Data Science, or a 'software engineer' to Software Engineering, and what shapes the expressions of their sociotechnical regimes?

Asignification, as it resides in Guattari's (2011) concept of an "assemblage of enunciation" (p. 45), plays a role in delimiting and shaping the signifiers expressed by a sociotechnical regime. Asignifiers play a role in signification greater than that which is belied by their axiomatization on a plane of reference, because they are inherent to any form of signification expressed by a regime: by consequently structuring and delimiting the forms of subjectivation used by a sociotechnical regime to individuate one identity type from another, they determine 'meaning' and 'value' both inside and out of a regime's boundaries. Guattari (2011) developed the concept to examine connections between "expression" and "content," which, he tacitly proposed, are "not attached to one another by virtue of the Holy Spirit" (p. 45). Assemblages of enunciation provide a basis—like a plane of reference—by which modes of "signification and semiotization" can hold or contain meaning relative to a state of affairs, a locality, an expression, or relation. In this way, conceptually a technical assemblage can be seen as the mechanistic exchange of data (e.g., "sign-particles") between abstract and concrete layers within a sociotechnical regime. These exchanges explain how Individuated subjects, like titular 'software developers' or 'mechanical engineers' are expressions and reflections of a 'dominant reality' (pp. 45-46), which gives weight to the signs "holding" (p. 45) the significance of their titles, and how those titles are themselves a reflection of a "certain mode of social organization" (p. 46). Thus identity, or a type of 'individuation' and 'subjectivation,' is a processual reciprocity

المنارة للاستشارات
www.manaraa.com

between 'asignification' and 'signification' in an assemblage of enunciation. Guattari created a set of tables, illustrating conceptual corridors for loose divisions between systems of signification and asignification, shown in Figures 4.1.

	Semiotic components	Functions of content	Semiotic components	Articulations of content and expression
Interpretive generative transformations	A. Analogical	Semantic	A. Analogical	Field of interpretance
	B. Semiological linguistic	Signifying	B. Semiological linguistic	Field of significance (double articulation)
Non-interpretive generative transformations	C. Symbolic intensive	Illocutionary, indexical and of passage	C. Symbolic intensive	Illocutionary, indexical and of passage
	D. Diagrammatic	Asignifying sense	D. Diagrammatic	Asignifying sense

Figure 4.1: Guattarian Assemblages of Enunciation (reproduced from Guattari, 2011, p. 57)

While the categories in Figure 4.1 are distinct, e.g., 'Diagrammatic' or 'Analogical,' Guattari (2011) admitted that any one category tends to expand and seep into its neighboring categories, because "[certain] contents are dominated by redundancies of resonance, others by redundancies of interaction" (p. 54). This means that "[an] assemblage of enunciation" is "derived sometimes from the side of signification and sometimes from the side of diagrammatism depending on the

المنارة للاستشارات
www.manaraa.com

transformations of its composition." An assemblage of enunciation comprises all the semiotic components in interpretive generative transformations and non-interpretive generative transformations; the enunciation itself is a materialization of the "fluxes, the territories, the machines, the universes of desire" of a "plane of consistency," which account for how "different ways of existence of systems of intensity do not spring from transcendental realities but from real processes of generation and transformation" (1984, p. 290). The idea is that expressions an assemblage of enunciation produce are always, in some way, of a 'mixed semiotics' which come to affect the world; the significance of asignification here is that utterances or expressions are in some way planned, are diagrammed and enacted. Deleuze and Guattari (1987) used the word 'diagram' to describe an open process by which to find new lines of flight in, essentially, a capitalist axiomatic, to use Guattari's language, that individuals could use to find new lines of flight and means of escape from otherwise despotic forms of subjectivation: "Connect, conjugate, continue: a whole 'diagram,' as opposed to still signifying and subjective programs" (p. 161). And here, they explain that assemblages can be tilted and made to "pass over to the side of the plane of consistency," thereby yielding new expressions. Diagrammatism and asignification are as important as signification in the mixed semiotic systems espoused by Guattari, such that diagrammatic processes are how expressions—enunciations—emerge upon a plane of consistency, at a certain point in time and space, and are actualized, materialized. For sociotechnical regimes, diagrammatic processes allow plans of subjectivation to be formed and imposed upon the subjectivated, which is a way of demonstrating Kittler's assertion about media determining human situations, or McLuhan's recognition that humans, woven into systems of automation, produce something new.

www.manaraa.com

Diagrammatism in McLuhan and Kittler: Consistency, Automation, and the Digital

While Guattari's arguments and definitional work about asignification—especially considering his *Machinic heterogenesis* essay (1995)—places him closer to Kittler than McLuhan in terms of the role of the human in machinic assemblages, in that humans are generally a product of their tools, it is difficult to escape the role of reciprocity inherent in diagrammatic processes. Genosko (2014) noted that

> decentring [sic] human subjectivity for the sake of machinic proto-subjectifications is one of the broad theoretical goals of Guattari's philosophy. The field of asignification becomes for Guattari that of non-human enunciation in and among machinic systems: strictly speaking, 'equations and plans which enunciate the machine and make it act in a diagrammatic capacity on technical and experimental apparatuses.' This vast region includes everything from machine language 'fetch and execute' routines, to system interoperability at different levels of exchange, or to multi-levelled cybernetic loops. These are scientifically formed by computer scientists and systems engineers. (pp. 17-18)

The significance of Genosko's interpretation of Guattari's arguments for asignification allows machines to speak, in the sense that they communicate and mediate human interactions. However, 'multi-level cybernetic loops' implies a reciprocating process, one which 'loops' from some point to another point, governed by responsive logics that respond according to feedback they are designed to receive. McLuhan (1994), in *Understanding Media*, firmly located the human at the center of 'cybernetic loops'; the human brain was the center of any reciprocating circuit. That machines now communicate independently of humans—at least on the surface—indicates a different and purely machinic layer of depth to the kind of automation he proposed, in that electricity and power are being used independently of human agency. But, as Guattari

المنارة للاستشارات
www.manaraa.com

argued, due to processes relying on asignification, many things have *always* been independent of human agency. Guattari favored soil metaphors for describing asignification and signification, because soil not only implies that they exist alongside one another in mixed proportions (Genosko, 2014), but describes how information does not need signification, and can be transmitted independently of systems of signification. As such, asignification is central not only to technological diagrammatization, like the work of 'computer scientists and systems engineers,' or of mechanical engineers designing apparatuses for specific material tolerances and intended uses, but to the emergent diagrammatization of machinic ontologies in general, because they *must* exist to both interfere with and enable the transmission of signals and meaning to and from virtual and material strata. For example, the systems of DNA, RNA, and mRNA function independently of any intent, but adhere to a pre-signifying diagrammatic which governs the way nucleic acids pair off or unwind: the bonds between acids, as they emerge, are what they are *independent* and *apriori* of any signification. Systems of asignification can be both intentional and designed, or emergent and pre-signifying properties of a system of reciprocal exchanges, e.g., the tests and measures used to evaluate the efficacy of a function or axiomatic (set of functions as praxis) in response to a proposition.

One thing that binds Guattari, McLuhan, and Kittler together in a way that helps conceive of diagrammatic processes as reciprocal praxis within a sociotechnical regime is the concept of the feedback loop, be it human, nonhuman, or cybernetic. McLuhan and Kittler's definitions of 'cybernetic' differ in terms of 'hopefulness': the timbre of McLuhan's perspective about the symbiotic role of technology in human life leads him to assert that the consciousness of an artificial intelligence "would still be one that was an extension of our consciousness, as a telescope is an extension of our eyes" (1994, p. 351). Kittler, on the other hand, with his

المنارة للاستشارات
www.manaraa.com

sardonic-but-entirely-serious quip about the "so-called Man" (1999, p. xxxix), views the human as a product of the telescope, or the human as a product of the artificial intelligence, because of the ways those technologies mediate and "determine our situation." Guattari certainly sees the human in terms of "subjectivation," such that even systems of values, e.g. "religious, aesthetic, scientific, ecosophic," become incorporated into the machinic systems which are "not only cybernetic feedback" in terms of organic humans, but as the assertion of a system's becoming, its heterogenesis (p. 54). But While Guattari decenters the human subject from machinic ontologies, he shares a peculiar ontological commitment with McLuhan, in terms of autopoiesis. A crucial argument McLuhan (1994) made to support his claims about the recursive nature of mediums had to do with 'automation technology,' which "creates roles for people, which is to say depth of involvement in their work and human association that our preceding mechanical technology had destroyed" (pp. 7-8). Automation is information, and as such, requires a type of learning and involvement that is always 'on' and fully immersive; for entertainment purposes, automation creates "mass media," whereas for industry automation causes "scientific revolution" (pp. 346-347). The way McLuhan wrote about automation demonstrated the difference of hot and cold mediums, which were 'shallow' and 'deep' in terms of how individuals participated with them. Television was a 'cool' medium because it relies on and creates "depth structures in art and entertainment alike," which lead to "audience involvement in depth" (p. 312); viewers were involved with cool mediums in ways that required electricity and the servomechanism to connect—neurologically—one person with an assemblage of machines. The "cybernation" McLuhan discussed was not only the operation of "electric energy" being "applied indifferently and quickly to many kinds of tasks" (p. 350), but emphasis of the centrality of 'human' to all layers of interaction of a system where information transmission was instantaneous (and perhaps

المنارة للاستشارات
www.manaraa.com

unambiguous). The de-centered human Guattari sought recognized the role of entropy and noise—of chaos—in the messy sedimentary layers of asignifying and signifying systems, whereas McLuhan ideally saw automation as an extension of human centrality and stability, which is more tenable in an orderly state of affairs.

While McLuhan and Guattari each had different attitudes toward technology, they both recognized similar ways of encoding information into horizontal, or flattened, topographies that entangled human nodes in one-to-many, and many-to-many relationships of signification and asignification. Both Guattari and McLuhan are interested in what new expressions a system can yield. At one level there is a virtual, abstract universe of theory, and then there are the actualized utterances of that theory in the world; from these abstract topographies of encodings a thing emerges, be it human subjectivity, a form of subjectivation, or a printed circuit board. Guattari developed the idea of 'consistency' as a way to "precisely define an assemblage in relation to its components and the fields in which it evolves," so that we could gain a sense of its "existential" qualities (p. 47). The concrete expressions of an assemblage—the product of subjectivation of a sociotechnical regime, for example—are premised in a multiplicity of ways. Specifically, their consistency relies broadly on the idea that their expression at a specific point in time and space emerges from a combination of molar, molecular, and abstract fields: Guattari uses particles in "contemporary physics that are 'virtualized' by a theory that only preserves their identity for a negligible time" to demonstrate how, as a "diagrammatic effect" a particle is both concretized expression of matter and the product of abstract displacement. As seen in Figure 4.2, Guattari used the idea of 'resonance' to imply signifying interactions—in this case, signifying fields—and classified "machinic redundancies of interaction" as the realm of sign particles:

www.manaraa.com

CONSISTENCY

	molar	molecular	abstract
redundancy of resonance	signifying fields	semantic fields	capitalistic abstractions
machinic redundancies of interaction	stratified fields	components of passage	constellation of sign particles (abstract machines)

Figure 4.2: Guattarian Fields of Consistency (reproduced from Guattari, 2011, p. 51).

Automated interactions, or "machinic redundancies of interaction," rely on a combination of stratified fields at the molar level, components of passage at the molecular level, and constellation of sign particles at the abstract level to express themselves (either virtually or concretely) (p. 51). The word "redundancy" is important to unpack, because not only does it resonate with McLuhan's automation, itself premised on new bodies forming from deep connections to and from wide topologies of interaction, but because it operates in the molar domain, where "elements are strongly crystallized and stratified, allowing flows of redundancies to develop" like "signifying effects" or "surplus value[s] of stratified codes" (p. 47). Due to how Guattari uses the word, redundancy implies repetition, similarity, and self-affirmation: a 'redundancy of resonance,' as shown in Figure 4.2, would mean that, due to capitalistic abstractions, e.g., the nature of capitalism itself, the connection of a signifier to its semantic content is truncated by an unnatural imposition, which becomes accepted because its implementation in an assemblage is inherent to many components across its strata. Guattari confirms this usage by explaining how capitalistic abstractions are "a cornerstone of signifying

www.manaraa.com

resonances and semantic fields" by operating at a "lethal level of abstract machinisms … which does not model the universe of representation" (p. 49). To make something new, one must create "new machines of diagrammatic signs-particles" to interfere with "semiotic fields and capitalistic abstractions" (p. 50). Consistency, then, provides the basis by which an expression of a system—like the new human for McLuhan—is made possible by the interaction between the signifying and asignifying planes of an assemblage of enunciation.

Diagrammatic Processes, Digital Technology

Systems of asignification underlie diagrammatic processes, and Guattarian diagrammatism explains aspects of McLuhan's and Kittler's definitions of *digital* technology for media studies. While systems of asignification like diagrammatic processes are actualized in nature—for instance, the expression of a seed growing in soil according to its immanent genetic components and facilities intersecting minerals appropriate to its germination—technology is an example of humanity co-opting asignifying principles for its own uses. As Watson (2008) explains, Guattarian diagrammatics are not only asignifying, but represent "processes of recording, data storage, and computer processing" (p. 12), all of which function with binary thresholds: like the seed sprouting or dying according to the presence or absence of a vital mineral at sufficient quantities in its soil, 'recording, data storage, and computer processing' not only require the presence of electricity, but further inscribe upon media an actualized binary state. Meaning does not, at the level of digital information, signify anything; rather, it reflects the presence or absence of a signal exposed, referentially, to an axiomatized plane.

Digital information in media is meaningless, at the semiotic and semantic levels, without first referring to the technical and social practices used to mobilize and deploy a plan of action

www.manaraa.com

(diagram) to use the correct interpretations (axiomatics) which allow 'meaning' to be extracted from that media. Axiomatics can be argued to be a type of diagrammatic effect, in the Guattarian sense, by providing the means to order noise, the functions which transform the noise into asemiotic responses to propositions, which then—through a kind of poeisis—yield their results to *usage*, to *utilization*. The level of signification intersects with processes of asignification at points where diagrammatic processes intervene in the human world (notwithstanding machine 'intelligence' capable of interpreting metaphor, thereby confusing itself). The process of referral provides a basis for—in programming terms—strongly typing a set of signals such that they can be recognized, individuated, as one type of data versus another. The metaphor of 'typing' is helpful for understanding Guattari's (2011) concerns with the integration of the computer "into the complexes of enunciation" (p. 103). He believed—or was at least concerned—that "it will become almost impossible to make a distinction between human creativity and machinic invention" (p. 103); typing, as a process of individuation at an asignifying level, allows us to deflate the idea of 'machinic invention,' which is the tuning fork by which Guattari's arguments resonate with Kittler's (1990; 1999; 2010; Kittler & Grumbrecht, 2013).[5] Kittler (2010) asserted that "[there] are no longer any differences between individual media or sensory fields: if digital computers send out sounds or images, whether to a so-called human-machine interface or not, they internally work only with endless strings of bits, which are represented by electric voltage" (pp. 225-226). At the level of the digital computer, this is true: at its most granular, asignifying level, all digital information processed by computers are 'endless strings of bits,' as Kittler states. But he conflates 'individual media' with 'sensory fields,' which does not make sense until

[5] As Winthrop-Young (2011) explains, Kittler saw the cultural strife during the 1960s as expression of "discursive and technological regimes which create and shape the so-called subjects who remain blissfully unaware of what makes them speak, think, and protest" (p. 25), which implies machines speaking through humans.

المنارة للاستشارات
www.manaraa.com

the introduction of a diagrammatic effect on media: affects and precepts distinguish themselves and are subject to interpretation by means of the presence or absence of a *type* of signal (Deleuze and Guattari, 1994).

Digital information, to transcend to a *type*, must satisfy diagrammatic requirements before it can be processed by a series of axioms, and only after its conformance to those requirements are validated is ‘data’ individuated as ‘video’ or ‘audio’ or ‘database’ or ‘text.’ Just as “sound” requires the presence of a continuous, analog waveform to be expressed as its type and satisfy the diagrammatic requirements of an auditory sensor, data—to transcend to a type, to become more than noise and escape entropy—must satisfy a set of diagrammatic requirements before it can be individuated into an ordered form. At the level of bits and bytes, data is meaningless noise until it is transformed by diagrammatic processes into states that can be used and employed, either to further other diagrammatic processes, as the presence or absence of some kind of result (in a discretely logical sense), or through the transformation and marshaling of that noise into something of *significance* for human interaction (i.e., MP3 audio files, or the probability that one political candidate will win over another).

Machinic invention seems to be premised on the idea that the creation of a thing—a new axiom perhaps that is used to affect the shape of a material outcome, like a new computer microchip—somehow hides the intervention of the human from its ontological telos (the factors that lead to the axiomatic realization) and is difficult reasoning to follow or accept from the perspective of art, philosophy, and science that Deleuze and Guattari (1994) argued. The axiomatics of the plane of reference used to create the digital computer were not only iteratively improved but refined over time based on humans observing the application of those axioms to the actualization of things. Planes of *reference* refer to the states of affairs from which they

المنارة للاستشارات
www.manaraa.com

emerge, and therefore have material and social components informing the practices (axioms, like problem-solving approaches) and the propositions and functions which they incorporate; planes of reference rely on reciprocal referents. The residence of an axiomatized function upon a plane of reference corresponds to the positive abilities of a thing's actualization to further improve its future axiomatization, which is crucially the materialist, empirical epistemological and ontological values of science. Kittler's assertions about the 'recursivity' of technics had more to do with the idea that the development of technologies evolves over time through reflexive practices. Recursion, from a software developer's perspective, does not hold the same meaning: recursive functions call themselves, and the same function calling itself is closed to revision. So, it does not follow that recursivity, in the borrowed-programming parlance he affects, creates change beyond the results the recursive function was bound to produce. Functions improve through *iteration* and *versioning*: work that is visited multiple times is said to be iterated upon; work that reaches a milestone, such as being feature complete according to a design document, becomes a new version.

Deleuze and Guattari (1994) recognize the value and limitations of numerical sets: axioms, at a certain point, produce entirely expected results due to, in mathematical terms, the predictability of their outcomes in accordance with set theorem and limits. The functions employed by axioms change according to material encounters in a state of affairs: "[functions] are, first of all, functions of states of affairs and thus constitute scientific propositions as the first type of prospects: their arguments are independent variables on which coordinations and potentializations are carried out that determine the necessary relations" (p. 155); and further, "[the] differences between the physico-mathematical, the logical, and the lived also pertain to functions (depending on whether bodies are grasped in the singularities of states of affairs, or as

المنارة للاستشارات

www.manaraa.com

themselves singular terms, or according to singular thresholds between perception and affection)" (p. 157). The propositions and functions that express the 'meaning' of source code or the design of microprocessors reside in a technical regime that is, in its own way, dominated by systems of signification, because they cannot be extracted from the state of affairs in which they reside. Expressions of diagrammatism, at the asignifying level, have 'meaning' relative only to a plane of reference: source code, for example, functions relative to planes of discrete logic, to syntax, to a compiler, to a microprocessor's architecture, whose expressions intersect at sign-points marking the presence or absence of a binary signal in response to requests for those signals. Signification is the domain of what *humans do* with the source code, when it executes on a computer (e.g., 'running' as a process) and affects change in a state of affairs, or in reading it for cultural meaning, analyzing it for patterns and practices reflecting biases, or for insights into idiosyncratic choices by a programmer. But at the level of asignification, source code is referential expression that satisfies diagrammatics which emerge from formal axiomatizations to solve propositions and functions in terms free from traditional significations.

For media studies, the issue with source code, and with any asignifying semiotic system, is that at some point their contents are translated into signifying semiotic domains, and that is where this process reciprocates. What makes Kittler's reasoning compatible with Guattari's asignifying and diagrammatic concepts is the recognition that raw data, which otherwise looks like noise, *can be made* to represent different 'sensory fields' *with sufficient processing*, which as Guattari (2011) states is how a computer can "deterritorialize sign machines so that the former end up acquiring a sort of asignifying transparency which perfectly enables them to be 'molded' in their techniques of representation and recomposition to the singular traits of matters of expression" (p. 104). So, when seen in a certain light, the ideas of the 'so-called man' or

المنارة للاستشارات
www.manaraa.com

decentered human that Kittler and Guattari espoused are attractive but are shone to be inconsistent with the premises of diagrammatization and the iterative behaviors and practices involved in the recognition, creation, and inscription of axiomatics upon a plane of reference. From the perspective of propositions, functions, and their state of affairs, a new version of an axiom—meaning an aspect of the axiomatic refers to a modified function on the plane of reference—is reincorporated into that function iteratively if the quantification and concretization of a thing is deemed ‘positive’ in accordance to another set of axioms designed to measure the thing. In this regard, Deleuze’s (1994) problematic focus on *difference* and *repetition* is suited to the analysis of planes of reference because it, combined with Guattari’s mixed semiotics, lead to the concept of transduction, which is a way to encapsulate the actions and consequences of intersecting groups of software engineers and customers.

Transduction is based on the behavior of electronic transducers, which translate a continuous analog value, like force, into a series of steps, which are discrete approximations within a minimum and maximum range of values. Transduction is ideally asignifying, because it involves a point of contact at which the translation of a natural encoding or signifying semiological value into a system of asignifying values occurs; and it is ideally signifying, because it represents a point of contact between an asignifying system where a translation is endowed with meaning. As a metaphor, it describes what happens when an agent of a sociotechnical regime groups works to negotiate two sets of problematics: their own, and their clients. The significance of Deleuze and Guattari’s theory for the study of software engineering is two-fold: first, their recognition that processes connect to other processes at many points, rhizomatically, across a plateau which allows source code to be seen as just one product of the many efforts contact between a software developer and their client generates; and second, the use

www.manaraa.com

of problematics to understand the impositions of competing ontological and epistemological commitments. Combined, it becomes evident that there is a problem inherent in communicating problems between developers and clients which is elucidated when one looks at how different their problematics intersect, are negotiated, and are understood.

Transduction

> Science is haunted not by its own unity but by a plane of reference constituted by all the limits or borders through which it confronts chaos. (Deleuze & Guattari, 1994, p. 119)

The previous section described asignification, part of Guattari's (1984, 2011) model of mixed semiotics, as fundamental to expressions generated by a sociotechnical regime. By comparing aspects of McLuhan and Kittler's discussions of technics, evidence for the presence of asignification and diagrammatism was found in their interpretations of machine and human interactions and scientific developments over time. This section describes how transduction translates, converts, or casts one type of semiotic component into another. It starts by defining what transduction is, looking at its conventional definition and examples in Guattari's and others' works, and turns to an analysis of the term in the framework of limiting fields and sign-points offered by Guattari (1984, 2011); it then examines how the discrete portion of transduction posits the necessary imposition of limits on a continuous flow of values as a basis for translation; and finally it explores how signification arises from asignifying systems using the emerging Data Sciences' discipline as an example. This section works to establish the signifying aspect of my exploration of the asignifying and signifying interactions at work in software engineering.

www.manaraa.com

Transduction is a helpful metaphor for understanding and describing the consequences of a double articulation of a problematic. Deleuze and Guattari's (1994) assertions about the design and implementation of a plane of references in conjunction with a state of affairs, especially in terms of how semiotic content is translated and made to become asemiotic content, and vice versa. But transduction, more importantly, sets up a means for understanding how iteration revises the axiomatics on planes of reference when signifying signs, the natural product of human interaction, get routed through diagrammatic processes that convert them into asignifying signs. Transduction also provides a means for understanding how functions and concepts intersect—and why they must, as Stengers (2005) observed, intersect "only after each has achieved its own specific self-fulfillment" (p. 151). The term "transducer" describes an electronic device that "converts variations in a physical quantity, such as pressure or brightness, into an electrical signal, or vice versa" ("Transducer," Oxford Dictionaries). Examples of transducers are found in smartphones to translate forces, such as acceleration, into quantifiable values that can be acted upon by software: the Apple iPhone uses an accelerometer (the "CoreMotion" accelerometer) to determine when the orientation of its screen is downward or upward facing along a Z-Axis (Jason, 2016; Apple, "Core Motion," n.d.), allowing its software to enable or disable its screen relative to how it is held by its user. The principle operation of a transducer is to convert one type of signal into another, and is therefore not concerned, in its actualization, with the 'meaning' behind its conversion. Its role is rather to fulfill the "first [functive]" requirement of defining "the limit and the variable" and the "relationship between values of the variable … with the limit," so that its imposition into the actual is always *in relation to* its axiomatized inscription on disciplinary plane of reference (Deleuze & Guattari, 1994, pp. 118-119). Transducers are great devices for demonstrating how science "relinquishes

المنارة للاستشارات

www.manaraa.com

the infinite, infinite speed" of philosophical concepts to "gain *a reference able to actualize the virtual*" (emphasis maintained, p. 118), because until the imposition of a limit upon a defined variable, force has no 'value' and is virtual, continuous, and uncoded energy. As it becomes a *function,* 'force' becomes axiomatized in that it has a set of associated practices that allow it to intersect with and work upon a state of affairs.

One of the principle assertions Deleuze and Guattari (1994) make about disciplinary "Thought-form[s]"—the demarcations between art, philosophy, and science—are that they each are products of respective approaches to 'chaos' (Arnott, 1999, p. 49). Chaos, or "the virtual as Deleuze was fond of calling it," was not merely a "lack of order," or even something seen as a "negative," but was treated "as a kind of pool of resources from which thought in its various forms extracts or 'clinches' new ideas, new ways of thinking" (p. 50). But the primary difference between concepts and functions, and the reason why they intersect only when fully realized in their respective thought-forms is that concepts are comprised of "inseparable components condensed" into themselves, while functions are "distinct determinations that must be matched in a discursive formation with other determinations taken in extension (variables)" (Deleuze & Guattari, 1994, p. 121). This means that, for science, limits and variables are derived from relations with a "state of affairs," which are "formed matter in the system" which may be "mathematical, physical," or "biological" in nature, such that the idea of "reference" itself is a manifestation of the "form of the proposition" (p. 122). Deleuze and Guattari argued that states of affairs are "functions," and are themselves "a complex variable that depends on a relation between at least two independent variables." Concepts relate to other concepts, are expressions of different conceptual "personae" (thinkers, like Descartes, Hegel, or Nietzsche), and intersect with other concepts across, potentially, many planes of immanence (p. 76). Concepts, on one

المنارة للاستشارات
www.manaraa.com

hand, collide in arguments of relevance or dominance with each other across personae and their respective planes (e.g., "philosophy always works blow by blow"), they do so at infinite speed. Functions, limits, and variables, on the other hand, are products and expressions of the slow speeds the sciences use to evaluate materiality. Speed is the ultimate bounds on a process used to determine, select, and evaluate functions: for science, time is essentially stopped to allow aspects of the virtual to be observed, whereas philosophy relies on infinite speed to allow for radical traversals of potential relationships between concepts upon their plane of immanence.

One aspect of the creation of concepts implies a type of thinking and activity that is relevant to a discussion of transduction and its scientific application and origination. "Laying out, inventing, and creating constitute the philosophic trinity," Deleuze and Guattari (1994) wrote, which are "diagrammatic, personalistic, and intensive features" (p. 77). This implies that a type of *asignifying* process is at work when "laying out" a plane of immanence: for the idea they represent, concepts and their components are mapped in such a way that their expressions are formalized so that they can both be referred to later through new traversals and intersections with other personae and planes of immanence. Is there a difference between philosophical and scientific information at the diagrammatic level? It could be that the organization and storage of information itself is asignifying—the ink and shape and form of character sets or script upon paper, the bits encoded into nonvolatile memory cells—but that the use information stored and organized in an asignifying system *becomes* signifying as it is translated into a signifying semiological domain.

Evidence that 'transduction' operates in the arguments and contributions Guattari brought to his collaborations with Deleuze is found in his work on collective assemblages of enunciation and "signs-particles" in *The Machinic Unconscious* (2011, p. 47). An 'assemblage of

المنارة للاستشارات
www.manaraa.com

enunciation' is not language itself, e.g., 'enunciation,' but the components that provide language meaning at a specific point in time relative to a place: "semiotization, subjectification, conscientialization, diagrammatism, and abstract machinisms" (p. 45); assemblages of enunciation operate somewhat independently from the "plane of content upon which they are inscribed," such that "their semiotic capacity for 'holding' a given subset of the world" depends on "their angle of significance in relation to the local conditions of the semiological triangle." Guattari's model of signification means that the meaning signs 'hold' are contingent on references made to "a worldliness," a manifold of components and their exchanges relative to a place in time and space, and that such relativity means that no "universal world" exists in which acts of enunciation produces the same meanings, the same signs of holding (p. 46). In an earlier essay, Guattari (1984) explained that "significations do not come from heaven, nor do they arise spontaneously out of a syntactical or semantic womb. They are inseparable from the power formations that generate them in shifting relationships of power. There is nothing universal or automatic about them" (p. 166). Following this reasoning, Guattari's (2011) assertions that not only do components within an assemblage of enunciation have to at some point convey signals and inert or actionable meaning to each other, but that such conveyance is a result of 'signs-points,' which are a "diagrammatic effect" makes sense (p. 47), because it provides a means of tracing transformations across "an interweaving of several such systems" as "a mixture of semiotics" (1984, p. 166). This implies that systems of asignification, e.g., diagrammatism and diagrammatic processes, inhere in, delimit, and govern assemblages of enunciation, which themselves are constructed relative to systems of power.

'Signs-particles' are the Guattarian mechanism of exchange between mixed semiologies of the real, the human, the abstract, the natural, the machine. They move and convey and relate

المنارة للاستشارات
www.manaraa.com

aspects of values or information—data—whereupon "abstract machines 'charge' themselves with redundancies of resonance (signification) or redundancies of interaction ('real' existence) depending on whether they are fixed and rendered powerless in a semiological substance or whether they inscribe themselves upon a machinic phylum" (p. 47). Machinic phylums are an area where Kittler and Guattari resonate: media determine our situation; "the web of history … is the machinic phylum," the aggregation of merging of the "military machine," "technological innovations," and "scientific revolutions" (Guattari, 1984, p. 121). To Guattari, the "machinic power of desire was, always and everywhere, already there" and thus, in a Kittlerian vein, determine our situation, always. And thus, our situations—our enunciations—are relative to proximate power structures. However, regardless of the synergy between these concepts, it is important to note that where Guattari (2011) is concerned, "all … assemblages of enunciation involving the human world are mixed" (p. 54), but that "abstract machines *are never definitively bound to fixed and universal coordinates*; they can always 'pull out' and re-emit quanta of possibility" despite being subject to diagrammatic processes and their effects (emphasis added). This implies that, where humans are concerned, transduction is a valid metaphor for examining what happens to signals and values in mixed semiotic systems, because something happens to a sign-particle (singular) or set of signs-particles within those systems that provides a means for a 'sign' to escape the referent 'the' effect a diagrammatic process seeks to impose (deterritorialize, overcode) upon what is ultimately produced by an enunciation.

Assemblages of enunciation supports the pragmatic idea of how identities and things are produced and inducted—transduced—by sociotechnical regimes by providing a mechanism for explaining how matters of expression are transduced to and from those regimes. Guattari (2011) mapped out three common *pragmatic* fields, demonstrating how matters of expression pass

المنارة للاستشارات
www.manaraa.com

through them (or into them) on their way to being assembled or *incorporated* expressions, which are seen in Figure 4.3 below:

	assemblage of enunciation	semiotic components	pragmatic fields
Field a	territorialized	icons and indexes	symbolic
Field b	individuated	semiological triangle	signifier
Field c	machinic collective	sign particles	diagrammatic

Figure 4.3: Three Limiting Fields (reproduced from Guattari, 2011, p. 60)

Matters of expression move to and from these fields: e.g., an expression 'produced' by a field moves from the assemblage of enunciation, to the semiotic components, and through the pragmatic fields before being expressed. This means that, to become *asignified*, to be encoded within a machinic collective, an expression passes through a diagrammatic process yielding appropriate sign particles which are compatible with that given machinic collective; something is taken apart and made to be compatible and inscribable when a basis for compatibility exists. Conversely, an asignifying expression, to be incorporated in an individuated assemblage of enunciation, must pass through a signifying process, becoming subject to the semiological triangle before being recognized as an encoded part of the given individuated assemblage of enunciation. This is important for the concept of transduction, because it demonstrates how Guattari was thinking of conversion at any stage of enunciation. Guattari proffered the notion of 'limiting fields' as a way to examine the production of an enunciation by respecting "the singular traits of each matter of expression" involved in its emergence (2011, pp. 60-61). Limiting fields are pragmatic and "result from the articulation of modes of encoding and from an inexhaustible formalization of a substance of universal expression" (p. 60). The purpose of these fields, which

المنارة للاستشارات
www.manaraa.com

are pragmatic in nature, e.g., “pragmatic fields,” or “pragmatic rhizomes” (pp. 60-61), are to coalesce matters of expression into semiotic action: they determine the construction of what is enunciated in a state of affairs. As a pragmatic field, a signifying or diagrammatic process is the real construction or deconstruction of an enunciation: a linguistic signifier, for the fact that it is expressed and concretized as audible utterance or inscription on paper or in the digital realm is indistinguishable in terms of its constitutive *process* from the “selection of the raw materials out of which it will be synthesized” used to compile source code into executables from electrical capacitances (DeLanda, 2010, p. 32). The reading and writing of source code, i.e., programming, can broadly be seen as what is *expressed* and *incorporated* by a sociotechnical regime in that it demonstrates how transduction is a process of what constantly determining what is and is not a ‘valid’ (e.g., legible) expression to and from the assemblage of enunciation incorporating the basis by which its symbols and signifiers codify meaning.

Necessary limitations

Transduction is an effective way to de-fetishize source code and programming by providing a theoretical basis for interpreting ‘code’ and ‘programming effort’ as a translating effect or form of mediation between different communicative and technical domains. As such, transduction is a succinct metaphor for describing how a sociotechnical regime grapples with an affirmative idea of chaos, as that which is not entropy so much as potentiality. In considering transducers as a scientific response to chaos, they become a way of thinking about interaction that excludes more than it includes in terms of what is evaluated and converted. Transducers operate processually upon the virtual by limiting potentials and giving form to a variable, so that its product, its output—the variable of its effort—represents only what it is designed to

المنارة للاستشارات
www.manaraa.com

recognize, within a set of thresholds. Transducers can also be thought of as points of intersection between concepts and functions, in instances where definitions of 'force' or 'momentum' reside in respective planes of reference and immanence. The application of diagrammatic processes to material encounters in the real world can be an act of reading, interpreting, and applying technical knowledge contained in specifications and trigonometry books to 'real world' problems of levers or rotational velocities and the avoidance of entropy; so, while concepts operate at infinite speed (Deleuze and Guattari, 1994), functions—because of transduction—demonstrably work to slow time down. As a type of process, transduction is used to convert the type of knowledge gained from observations of a function's deployment, diagrammatically, to real world problems to the iteration and versioning of the functions and propositions inherent to those diagrammatic process. This is the circular form of iteration which allows scientific sociotechnical regimes to advance. What remains to be seen, however, is how the episteme of a technical regime chooses to define its transducers, because diagrammatic processes, at some point, intrude upon and intersect with signifying processes. Through interaction, new meanings are rendered, concomitantly increasing semantic noise. Signifying processes—like the needs assessment discussions between project managers and customers—in turn, must produce a type of asemiotic knowledge that intersects and interfaces with diagrammatic processes (e.g., product specifications and requirements). In this way, a diagrammatic encounter in a world that is full of semantic 'noise,' from the perspective of asignification, can be stripped of signification so that an axiom, a function, or the shape and expression of a proposition can be revised, its asignifying map of connections iterated, or disconnected from other points on a plane of reference entirely.

The limiting of signification by a diagrammatic process gives rise to the idea of 'axiom' as algorithm, and transduction as the method by which such a process manifests itself in the

المنارة للاستشارات

www.manaraa.com

world, and how information is transmitted to and from the process. Transduction is compatible with Deleuze and Guattari's concept of "double articulation" (p. 40, 1987) which as DeLanda (2010) explains, is concerned with how materiality expresses identity (p. 32). Identity is important because, from the perspective of a plane of reference, it serves to index—like an ordinate—the function required by an axiom to respond to a proposition. DeLanda describes double articulation in relation to emergent principles: it "refers in the first place to material expressivity," to the properties that make matter "so dramatically expressive," and secondly, it refers to an articulation "that consolidates the ephemeral form created by the first and that produces the final material entity defined by a set of emergent properties that express its identity" (p. 32). DeLanda recognized that Deleuze and Guattari "used a variety of terms to refer to each of these two articulatory operations," and so defined the first articulation as "territorialization," and the second as "coding" (p. 33). Explained in turn, territorialization derives from Deleuze and Guattari's (1994) use the word 'territory' and its operators—territorialize, deterritorialize, reterritorialize—which both break down the subject/object dichotomy inherent in Kantianism, and explain how "thinking takes place in the relationship of territory and the earth" (p. 85). Such a move allows the act of material emergence to demonstrate a type of rhizome or assemblage so that the "earth is not one element among others but rather brings together all the elements within a single embrace while using one or another of them to deterritorialize territory." Hence thinking is not transcendent (Kantian), but is rather an expression of territory, which itself is from and of the earth, and is destined to become the earth at some point e.g., deterritorialization, "from territory to the earth," and reterritorialization, "from earth to territory" (p. 86). The earth can be thought of as a type of substance in the Spinozan sense. Territorialization, therefore, "concerns a *formed materiality*" (DeLanda, 2010, p. 33).

المنارة للاستشارات
www.manaraa.com

Limitations get at the heart of materialism undergirding Deleuze and Guattari's thinking by providing a basis of comparison (and therefore a type of stasis or compatibility) between different thought forms.

Guattari's distinction between natural encodings, asignifying semiotics, and signifying semiotics (Guattari, 1984, pp. 73-76; Watson, 2008) alludes to differences between natural expressions and asignifying ones, and of how diagrammatic processes, like an algorithm in software or a machinic process, have a formative role in the expression of a thing through the imposition of material limitations, inclusions, and exclusions. If materialism "must be understood … as a polemical position that combats any priority afforded to thought over matter, to mind over body, not in order to invert that relationship and give matter the same privilege, but rather to establish an equality between the two realms" (Hardt, 1993, p. 114), it demonstrates that there must be a basis for compatibility. Transduction explains why limitations are important by providing a basis for compatibility, for functions and propositions to relate to philosophical concepts, or for them to relate to percepts and affects. There is a stability implied by the imposition of limits on one or more values at the outset of a diagrammatic process, for to work at all, diagrammatic processes intersecting with 'the earth' must be able to convert it into 'territory' that satisfies its requirement for a recognized variable of a given type and within a specified range. But even limited, the conversion of something stored in and organized by a diagrammatic process yields a wild range of signifying meanings and interactions.

Potentials from limits

The massive object graphs of a machine learning system can be indecipherable, compared to the surprisingly intuitive insights into the data those processes produce. It is

المنارة للاستشارات
www.manaraa.com

tempting to treat the result as a fetish, because the number of objects at work in a machine learning system make it supposedly impossible for a human to inspect them all, to see all a system's inner workings. And yet, the system is designed and implemented, somehow, from principles—axiomatics—derived from experience, and for purposes related to problems. Machine learning should not be treated as a fetish; rather, the processes producing machine learning should be interrogated as a mediating effect, a mediating effect of a sociotechnical regime. To see software as more than a fetish for an unknowable It is easiest to show how signification arises from transduction by describing Data Science, an emerging subdiscipline of Computer Science. Despite the imposition of limits upon matters of expression, the pragmatic fields at work within Data Science are a cogently appropriate example of transduction because of its methods of selecting interesting (and signifying) potentialities from chaotic arrays of otherwise inscrutable data. Data Science is also, as of this writing, a valuable demonstration because it represents intersecting issues of the limitations of design inherent in software engineering, the expansive nature of signifying human practices, the reciprocity inherent in that connection, and the underlying expressive role of diagrammatic processes in a socius. Data science is a discipline emerging from Computer Science in the 1990s, which began as "discussions relating to the need for statisticians to join with computer scientists to bring mathematical rigor to the computational analysis of large data sets" (Kelleher & Tierney, 2018, p. 17). A central premise at the root of the *propositions* examined by data science is, "[if] a human expert can easily create a pattern in his or her own mind, it is generally not worth the time and effort of using data science to 'discover' it" (p. 4), and one of the clearest examples of functions, axiomatics, and propositions can be read out of a description of how data science has moved away from relational databases (using tables and columns and views that look like Excel

المنارة للاستشارات
www.manaraa.com

spreadsheets), and toward object representation, where data is stored with its attributes (values like the 'subject' line of an email, the time and date it was sent, its author, etc.). The move to object representation (where attributes become, for analytical purposes, the salient aspect of the data) from relational databases (which rely on primary keys, which are special columns of data that store a unique identifier, like a 'customer ID' value in an 8-byte unsigned integer value) arose because formal, schematized data types mapped across columns and tables impose too much overhead on the probabilistic analysis of values for similarities.

Relational databases (e.g., Simple Query Language (SQL) databases like Oracle or Microsoft SQL) are great for storing data that can be easily schematized, and provide an efficient means for storing and selecting data: when a bank looks for a customer's transaction records for a particular day, it queries a relational database for any transactions appearing on any of the payment source identifiers (a checking account number, a savings account number, a bank card, a credit card, etc.) relating to the identifier the bank assigned the customer. While the strength of relational databases resides in their ability to "store data in tables with a structure of one row per instance and one column per attribute," which is "ideal for storing data because it can be decomposed into natural attributes," the problem with relational databases became apparent to businesses and computer and data scientists because "the amount and variety of data generated by different parts of [a company] have dramatically increased" over time (p. 8). Traditionally, a software engineer might derive a database schema, which is the map of tables, columns, and data types the database will use from the needs assessment and requirements gathering processes. After the schema is written, it will be deployed, becoming the database for which the tools the software engineer was asked to implement will be written. It is costly and troubling to change a schema once it is deployed, because changes to a schema require changes and testing of the tools

المنارة للاستشارات
www.manaraa.com

that were written for and rely on that schema. In a Deleuze and Guattarian sense, relational databases are great for storing known data, but make it difficult to capture *emerging* data, new types of relations that are immanent to the experiences inherent in the schema, because a schema requires everything stored by it to conform to it.

The conformity of schematized data comes with a certain amount of computational baggage. The problematics of data science have yielded the production of new functions, axioms, and diagrammatic processes to make sense of exponential data growth. A core *problematic* data science tackles with is an ability to produce “actionable insight” about a “problem that enables us to do something to help solve the problem” by looking at “patterns” across potentially “millions of attributes” (pp. 4-5). Insight requires clarity, or a view of data that is manageable and efficient, computationally. In a scenario where a person might have an identifier that can be used across many systems—like a Social Security Number (SSN) in the United States—relational databases have the (somewhat ironic) benefit of allowing themselves to be related *to other* databases in other private companies, state agencies, and federal institutions. This produces an odd entanglement with software engineering, because such connections cannot be foreseen, and when schemas across databases are joined, they essentially produce new databases, for all intents and purposes: designers and developers cannot anticipate and test for every conceivable connection. The software written for such interconnected systems becomes logarithmically difficult to maintain and test in terms of human labor. Enter data science, which has now largely shed relational databases in favor of object representation databases, like “NoSQL,” which allows data to have a “flexible representation” because “attributes for each object [are] encapsulated within” it (pp. 9-10), and rather than attempt to enumerate all possible interpretations, the most common analytic frameworks, like machine learning (ML), use software to analyze data to “[build]

المنارة للاستشارات
www.manaraa.com

accurate prediction models" (p. 18) that provide insight into the data, therefore allowing some action to take place. As Alpaydin (2016) explains

> Machine learning, and prediction, is possible because the world has regularities. Things in the world change smoothly. We are not "beamed" from point A to point B, but we need to pass through a sequence of intermediate locations. … The ability of generalization is the basic power of machine learning; it allows going beyond the training instances. Of course, there is no guarantee that a machine learning model generalizes correctly—it depends on how suitable the model is for the task, how much training data there is, and how well the model parameters are optimized—but if it does generalize well, we have a model that is much more than the data. (pp. 40-42)

Alpaydin's explanation alludes to the idea that an emergent aspect of data is present in machine learning: at some point, a well-generalized predictive model will tell us about the rules and common errors inherent to a process, especially as it relates to the idea of continuity, e.g., 'smoothness.' The data used by a machine learning model—the self-contained NoSQL row—is the concretized expression of a diagrammatic process used to fit matters of expression into place, such that the model moves from one point to another. Pragmatically, the row cannot contain everything, so the diagrammatic process constituting it sheared away irrelevant information; the billions or trillions of rows a machine learning system uses contain information deemed relevant by the mitigating and limiting effect of the pragmatic field governing their expression as data. At the level of rows and columns, the data is meaningless to humans, because its meaning becomes apparent only when a machine learning system enunciates its totality, in aggregate.

Transduction conceptually allows us to embrace the role of diagrammatics and asignification in producing signification. Continuity is a product of confluence; smoothness, in

المنارة للاستشارات
www.manaraa.com

computation, is an illusion produced by an abundance of resources. Alpaydin's explanation of what makes machine learning possible is an important intersection with Guattari's work on diagrammatics and pragmatic fields, because a well generalized model "that is much more than the data" is a product of trillions of calculations that rely on trillions of transducive actions used to prepare the data for the machine learning system. For, despite things in the world (i.e., nature, of earth, of analog and continuous interactions, of geographic striation) seeming to smoothly move or transition from one point to another, "we need to pass through a sequence of intermediate locations" (Alpaydin, pp. 40-42). 'Data,' in the abstract sense of the *computational* term, is always an expression of transduction, the product of transducive action. Guattari inherently recognized this, or, at the very least, produced a basis for recognizing the role of diagrammatic processes in the production of data, of the shedding of signification so that a type of compatibility between a symbol or signifier in the world could be made to work at the level of an asignifying process. But conversely, transduction is a point of analysis which allows us interrogate intersections where diagrammatic expressions *become* signifying, where semiotic processes incorporate asignifying values through human interactions.

It is in those interactions, in the necessary imposition of limitations on signifiers to understand them asignifyingly, that software developers struggle through a double articulation of their own problematic. Before they can understand their clients' or users' needs, they must understand how to articulate their own sociotechnical regime's diagrammatic processes in a way their clients' or users' can understand, which alludes to a problem of double articulation. On one hand, the double articulation represents a developer explaining their reasoning, based on asignifying principles, in a way that can be understood by their clients, who may not have access to the developer's plane of reference; and on the other, the client must understand their problem

www.manaraa.com

well enough to explain it to the developer, so that the developer can, by parsing the client's signifying language, align what the client means and wants to an appropriate diagrammatic process capable of producing what they want.

Software engineering and the development of software is emblematic of a constant negotiation of mixtures asignifying and signifying semiotics in software's becoming and invested knowledges. This chapter began with the task of describing and defining the diagrammatic, asignifying parts of Guattari's mixed semiotics in relation to software engineering, and looked at how asignifying, diagrammatic encodings or inscriptions work and are ordered upon the technical discipline's plane of reference. Next, this chapter introduced the concept of transduction to describe a liminal means of moving values into and out of that plane of reference, which operates by approximation, by the shedding of excess meaning (signifiers) to fit values into diagrammatic processes. Now it turns toward the other side of a transducive process, to explore the realm of signification, which encompasses human interactions generally. When problem-solving, software engineering tends to work from 'best practices,' which are axioms of an idealized, diagrammatic form. This section explores the implications of the signifying realm software engineers negotiate when they attempt to transduce a problem and its conditions from a customer, which can be an individual, a corporation, or even an imagined, idealized user.

Communicating problems so that they are understood is a problem in and of itself, and the task of aligning signifying values to diagrammatic ones is prone to misunderstandings. Signification produces meanings derived from usages and contexts, so software engineers receive and parse the transmission of their clients from a Weaver-esque semantic noise. After having seen how a type of diagrammatism operates within theories of automation and recursion

المنارة للاستشارات
www.manaraa.com

at the level of *technics* in McLuhan and Kittler's work, it is now possible to examine a real axiomatic, e.g., the concept of the 'best practice,' and its interactions with a signifying world of human interactions. The purpose of this section is not to redefine the concept of signification or the signifier, but to both show how it operates in a technical process, from a Deleuze and Guattarian perspective, so its role in the actualization of diagrammatic engineering praxis in software might be elucidated for other forms of technical inquiry in media studies.

This section begins by defining signification so that it can be constrasted with asignification, clarifying its role in a transducive process. This allows us to understand how they differ, in Guattarian terms, and to value the trend of signifying systems to subject (personify, etc.) non-signifying systems (like natural encodings, musical notation, computer code). I then turn to the issue of the 'noise' inherent to semantics, and of the role of 'best practices' in software engineering as attempts to mediate and simplify the acts of interpreting a singular meaning from contexts in which multiple meanings reside, and how these practices continue to be error prone. Finally, I explore how transduction brings problematics into conflict, and how the act of understanding a problem through a diagrammatic lens like a best practice, in the case of software developers interacting with customers (real or imagined), is a negotiation of multiple double-articulations of the conditions of different problems. This section highlights the role of noise/semantic noise and the inherent difficulties of transducing signifying signs into asignifying signs and vice versa, of why software might be so difficult to make, and how Deleuze and Guattari's theory and the framework of the sociotechnical regime offers a sophisticated and nuanced means to interpret the processes of software engineering.

Signification for Deleuze and Guattari is not principally linguistic in nature. Colebrook (2002) described Deleuze's definition of signification as principally empirical, where "all life is a

المنارة للاستشارات
www.manaraa.com

flow of signs; each perception is a sign of what lies beyond, and there is no ultimate referent or 'signified' that lies behind this world of signs" (p. 86). Deleuze resisted the primacy of language and its ability to "act as a privileged and independent subject or agent" in structuralist definitions (p. 107), and instead argued for a kind of signification that is productive, rather than representative: genetic codes work through life, producing it as an expression of immanence, in the sense that those codes exist outside of human minds whether imagined or not. The scope of the sign, for Deleuze, expands beyond language to incorporate many types of codes. Guattari (1984) refined signification in his own work, incorporating an understanding of signification that recognized flows of codes affecting the becoming of utterances or expressions of matter by arguing for three co-related types of semiotic channels, namely *non-semiotic encodings*, things like genetic codes which act "independently of any semiotic substance"; *signifying semiologies* which are "based upon systems of signs" and have a "relationship of formalization on the plane both of content and expression"; and *a-signifying semiotics*, which are things like "a mathematical sign machine not intended to produce significations" (pp. 74-75). Just as Deleuze resisted privileging language, Guattari warned semiologists to "avoid the semiotic mistake of projecting the idea of 'inscription' onto the world of nature," for there "is no genetic 'handwriting'" (p. 74). Non-signifying semiotics "do not produce effects of meaning" in the way signifying semiotics do because they can enter into "direct contact with their referents," i.e. points-signs (p. 290). Signifying semiotics produce meaning two ways: symbolically, through "various types of substance" like "gesture" or "ritual," which can never be fully translated into "systems of signification"; and through signification itself, which coerces or replaces other forms of substance such that a "single signifying substance" replaces them (pp. 74-75). Signification is a systematized set of references that limit or exclude "all other poly-centered semiotic

المنارة للاستشارات
www.manaraa.com

substances" such that they become "dependent upon a single specific stratum of the signifier" (p. 75).

While signification is similar in many ways to asignification when considering the requirement that 'meaning' be systematized using arbitrary (e.g., despotic) means, signification differs from asignification primarily in terms of what their dependencies refer. For signification, some authority must connect a meaning to a sign, which excludes other potential meanings; Guattari (1984) explained that "[writing] machines are essentially linked to the setting-up of State power machines" (p. 75). The meaning of a signifying sign therefore relates to a process of power, without which it would lose meaning, which is of import for media studies:

> The effect of the written word in the unconscious is from thenceforth fundamental—not because it relates back to an archetypal written language, but because it manifests the permanence of a despotic significance which, through arising out of particular historical conditions, can none the less continue to develop and extend its effects into other conditions. (Guattari, 1984, p. 75)

Signifiers mean what they mean because of a form of authoritarian coercion that excludes other meanings (over-codes). The basis of 'the meaning' of a signifying sign refers to a despotic regime, and if that regime ceases to exist, those meanings become untethered, losing significance until they become attached to another despot. While functions on a plane of reference are placed there by a regime, which might be its own type of despot, their meanings depend on their ability to refer to something real. Functions can directly connect to a referent, to the "most de-territorialized material fluxes," which "operate independently of whether or not they signify anything to anybody." This means that only asignifying signs can act as point-signs, allowing connections from an abstraction (function) to matter (territory).

المنارة للاستشارات
www.manaraa.com

Despite being disconnected from 'territory' in the way asignifiers are, signifiers complete a circuit in the process of transduction by allowing point-signs to be imagined, discovered, and explained in language: signifying semiotics are inescapable. So, while asignifying "machines" like science, music, art, or computation ("analytic revolutionary machine") "remain based on signifying semiotics," they only use them like an "instrument of semiotic de-territorialization," because they "operate independently of whether or not they signify anything to anybody" (Guattari, 1984, p. 75). Essentially what delineates signifiers from asignifiers is the dependence of an asignifying sign on a concrete referent that does not "involve any relationship of superiority or subjection" to describe; the language literally spoken in and by sociotechnical regimes, at least where functions and propositions are concerned, effectively becomes asignifying because talking through functions frees it from the subjection of despotic, coerced signs. The 'meaning' of an asignifying code (or value) described in otherwise signifying language is the expressive property of a function given a related proposition, which allows a function to be explained in language in a way that proscribes domination by its authoritarian system. A computational example of this is the toggling of a bit in a memory cell: it fulfills its asignifying role by signaling the presence of a value to a broader system, but its meaning resides elsewhere. This may be an example of language in despotic systems yielding to the requirements of diagrammatic or asignifying expressions to facilitate their enunciations. If so, this explains how planes of reference are not only organized according to matters of efficacy, but for the ability of their functions to be expressed in language that ideally resists the domination of signifying systems. In a sociotechnical regime where functions are valued according to their ability to respond to propositions in the pursuit of solving specific problems, it follows that negotiating the signifying language of the problem and its transmittal into a regime is a point of

المنارة للاستشارات
www.manaraa.com

problematic intersection: the problem may be related to natural encodings, or an asignifying expression, but attempts to *describe* a problem's conditions are subject to semantic noise and distortions of power when they move through human intermediaries.

Ideally, the axiomatics of planning and designing solutions, often called 'best practices' by software developers and project managers (Jones, 2010), offer a clear map showing all of the steps needed to produce a solution from beginning to end. If an algorithm directs data through sequences of computer instructions, which are themselves (ideally) predictable, reproducible, and efficient steps, a best practice is like an algorithm for software engineering in that it might offers a route "that can be followed from the very beginning of a software project all the way through the development" leading "to a successful delivery" (Jones, 2010, p. 10). While software engineering does require its practitioners to have "state certification, licensing, board examinations, formal specialties" and so on, Capers Jones recognizes that "neither standards nor certification have demonstrated much in the way of tangible improvements in software success rates" (p. 9). Best practices are sets of principles learned from trial and error and seem to be recognized as such when enough problems of a given type with similar conditions have been adequately solved, and ideally, they are flexible enough to account for conditional differences. However, attempts to manage the collective human effort required to marshal the shape, timbre, and content of software are subject to semantic noise and human idiosyncrasies and fallibilities.

The human ideal of 'clear communication,' from the perspective of a problematic, is impregnated with fallibility for historical reasons: in Chapter 3, for instance, human fallibility was evidenced by Brooks and IBM's travails to develop OS/360, hampered by schedule slips caused by human illnesses, by idiosyncratic programming techniques, and exacerbated by varying degrees of interpersonal and technical skills. The issues IBM encountered developing

المنارة للاستشارات
www.manaraa.com

OS/360 conspired to drive the cost of the project into astronomical units of dollars and man-hours spent. Brooks' findings resonate today, evidenced by the title of Ronald Day's (2017) work, *Design Error: A* ***Human Factors*** *Approach* (bold added) which explains, on the first page of its introduction, that while human designers "create new computer programs to do everything from making your microwave work to telling the different components of your car engine how to work together," design errors can lead to vehicles crashing, buildings falling over, and rockets carrying costly satellites into orbit disintegrating on launch (p. xiii). If designers create the "work processes and procedures that guide the people who get the job done," they also introduce errors into those processes which lead to failures of one kind or another: Day delineates, between human and design errors, e.g., implementation and process. If a diagrammatic process can be looked upon as a series of steps leading to predictable outcomes, it follows that the steps of a process should be evident, making decisions obvious based on predicating factors. Improper design, however, will at least increase the likelihood that a product, like software, will fail in some way.

Design errors and failures can often be traced back to inadequate understandings of a problem by those designers seeking to solve it. For human-centered design tasks, Day (2017) asserts that successful designs incorporate an understanding of the "human factors … surrounding person plus machine plus operational environment" (p. 11). Technical successes, e.g., "the design met the specifications," can still fail operationally by not being "user-friendly." If signifying semiologies "articulate signifying chains and signifying contents" (Guattari, 1984, pp. 289-290), it is evident that something interferes with the transition or translation of a value—like that which can be talked about amongst humans—into the model of the process that leads to the production of a product, such as software. While at the diagrammatic level it is entirely

المنارة للاستشارات
www.manaraa.com

possible to select the wrong function in response to a proposition, signifying semiologies play a mediating role in design processes because they are used to describe "problem-solving domain" in which a designer operates, which is the basis of knowing what prior solutions have been constructed and may be applicable within the current context (Day, 2017, p. 19). The process typically contains five steps, shown in Table 4.1.

Table 4.1: Typical steps for a design process (reproduced from Day, 2017, p. 43).

1. Forming a design concept
2. Describing that concept in a set of specifications
3. Building, engineering and writing the design product
4. Testing the design build
5. Implementing the design in the workplace

Of those steps, the first is the "most risk-prone," because not only has a problem arisen which compels a solution, but designers must "take into account every factor, every variable associated with a successful solving of the problem" (pp. 43-44). Design errors established in the first stage of the process can lead to the failure of subsequent stages, or to an inherently flawed product. Beyond failing to account for every variable, on one hand, designers implement errors in the first stage of work by failing to communicate with the users their solution incorporates: "[this] lack of contact with end users often means designers lack a clear view of the operation setting and their designs are not as suited to the task [of solving the problem] as they might be" (p. 44). On the other hand, "clients are part of the problem" (p. 12). Despite functions being adequately described in signifying language—because of their enunciation's inherent abilities to resist systems of signification—the 'best practices' (which Table 4.1 ultimately depicts) approach for

المنارة للاستشارات
www.manaraa.com

software engineering still moves in the domain of signification because of how one or more problems reside in the conditions its *users* cohabitate. While best practices attempt to make a problem recognizable by the conditions surrounding an issue and represent successful efforts to solve the problem at the root of those conditions, matching conditions to a problem (or type of problem).

Problematics and Transduction

Matching the conditions of a problem to the *correct* problem is at the root of issues of transduction. Deleuze's (1994) problem-orientation[6] can explain how a 'best practice' can be matched to a set of conditions. Despite best practices ideally governing the axiomatics of a problem-solving domain as an expression of a plane of reference, work is required to adequately match the problem underlying the 'best practice' to the set of conditions in which a software developer works. This matching of problems to conditions relies on reciprocating, transductive processes between asignifying and signifying domains, which makes it susceptible to failure (or Spinozan termed, inadequacy). For Deleuze, solutions to problems are principally mathematical, whether "physical, biological, psychical, or sociological" (p. 179), and while the mathematics of problematics Deleuze explains can be inscrutable at times, he describes the "universal synthesis of the Idea" as "the reciprocal dependent of the degrees of the relation, and ultimately the reciprocal dependence of the relations themselves" (p. 173), indicating that problems have a type of primacy in determining the relations that express things. The "complete determination of a problem is inseparable from the existence, the number and the distribution of the determinant points *which precisely provide its conditions*" (p. 177). Problems are inadequately expressed

[6] Ideological investments regarding 'false' problems notwithstanding.

المنارة للاستشارات
www.manaraa.com

when "a lack of understanding of the ideal objective nature of the problematic" occurs, which means its conditions are explained or understood in ways that reduce them "to errors" or "fictions." So, it is not only possible for designers to misunderstand customers, but for customers to misunderstand their own problems by failing to adequately account for their conditions. Such issues lead to solutions where the mismatch of one or more 'best practices' to a problem stems from inadequately understanding the conditions in which it resides and how it was communicated, resulting in efforts that fails to solve anything.

Historically it is evident that software developers tasked with gathering a client's requirements can misunderstand them, even in the presence of adequately described conditions. While Day's (2017) work offers a best practices approach to reducing design errors, it also overemphasizes the failures of clients to understand the "nitty-gritty of operations," or to not be "aware of human-factors issues" by implying those issues are principally client-side (p. 12). The reality is that the signifying side of transduction is prone to miscommunication and misunderstanding precisely for the reasons Deleuze explained, because while clients may work to explain experiential conditions—issues they are actively dealing with and seek to mitigate in some way—designers and software developers are always working to understand the problematic of their encounters with clients in addition to understanding the problematic the clients are explaining. This is precisely where software engineers embroiled in transducive processes, like needs assessments and requirements gathering, encounter their own problematic of a double-articulation, which is the act of adequately shedding the appropriate excess meanings and possibilities signifiers can contain so that compatible sets of signifiers can align and intersect with sets of asignifiers. An adequate understanding of a problem allows the appropriate functions to respond to or work on the correct proposition. The conditions embedded in a problematic

www.manaraa.com

bring signifiers and asignifiers into alignment, while the problem binds them together as an adequate set capable of offering a solution: the conditions of a problem determine the kinds of functions and propositions a solution will need to enact. However, despite the primacy of 'best practices' dictating design processes, they can only ever be guidelines when designers and clients intersect. Designers and developers must overcome the problematic of the double-articulation to adequately describe the client's problematic before they can even begin to solve the client's problem.

The best practices of software development and other engineering and design fields rely on a type of modularity, which can be talked through in the following way: given what is known about the problem and the context in which the solution must be expressed, the *best* practice for this type of problem *and* context indicates the following plan of action because of historical precedent. Unpacked and simplified, the prior statement becomes 'in similar situations these steps have been shown to work.' While modularity is the strength of 'best practices,' and are lauded for their reproducibility—much like algorithms—and their allowance for more substitutability of conditions and variables, they are themselves evidence for the problematic of a double articulation that was alluded to when Genosko (2014) described Guattari's attitude toward "information theory's "'skirmish' with meaning" (p. 13).

Best practices—and of the way's practices are axiomatized in a sociotechnical regime—can be seen as a mediating effect of a sociotechnical regime. At the root of a sociotechnical regime are sets of social and technical practices that are sometimes at odds with each other, due to the messy nature of signifying values, and the sometimes impractically precise reasoning behind asignifying values. At the core of good software is good design, which is born of experience, which itself is a repetition of encounters between a regime and the problems it seeks

المنارة للاستشارات
www.manaraa.com

to solve. De-fetishizing software and programming is enormously important for media studies scholarship because it avails a broader set of mediating effects into discussions about the nature of human and technical relations. The next chapter concludes this project by examining the outcomes of these chapters, while offering a way forward for future scholarship for the study of sociotechnical regimes and the media effects of problematics.

www.manaraa.com

Chapter 5: On Sociotechnical Regimes, Software Engineering, and Transducive Practices

This dissertation is a response to Wendy Chun's (2011) media studies' argument that software is a ghostly, fetishistic medium, which effectively cedes the responsibility of 'knowing' it by making it unknowable and ephemeral. Software is knowable if its media studies' definitions of the term are re-oriented around a Deleuze and Guattarian argument for self-sufficiency by working from scientific and industrial definitions of the term. From a Deleuze and Guattarian perspective, software is knowable if the scientific and technical definitions comprising it are respected, rather than over-coded by ancillary, philosophical ones. Working with scientific and technical definitions in a Deleuze and Guattarian way allows the many processes involved in producing software to be seen as propositions and functions on a plane of reference that interacts with a state of affairs independently of—but rhizomatically related to—philosophical and artistic concerns, which yields deeper understandings of the what's and why's behind software. No longer is it an ephemeral state of being, but an explicit one. Deleuze and Guattari (1994) argued that the thought-forms of science, philosophy, and art had no business interfering with the performance of each other's work by demonstrating how each thought-form required independence to operate. The thought-forms Deleuze and Guattari identified are effectively self-sufficient, because, as was shown in Chapter 2, the products of each thought-form only interact once they are fully mature, each created by their own means. Each thought-form has all that it needs to perform its work: this self-sufficiency provides a mechanism for philosophy to create concepts, for science to create functions and propositions, and for art to create percepts and affects for their own purposes and according to their own values. Scientific functions operate in a state of affairs; philosophical concepts might have no bearing upon a state of affairs but be novel and interesting enough to warrant their existence. Functions and concepts have different life

www.manaraa.com

spans, for instance: where at some point in time the function may be discarded for being inefficacious, the concept can find new life through a process of eternal return as it is reached for in imaginative ways across new philosophical plateaus. The key here is that the thought-forms act as *independent lines* upon a rhizomatic graph of relations that spans across virtual and actual horizons. So, while science *does not* create philosophical concepts, and philosophy *does not* create functions and propositions, at certain points in time their independent lines will intersect for a moment. These intersections give rise to new things, to new concepts and functions, because through their relationality they impact the thinking and imaginations of their participants, inspiring new 'becomings' by revealing lines of flight upon the relational graph that may have seemed impossible before. Deleuze and Guattari wrote that "philosophical concepts act no more in the constitution of scientific functions than do functions in the constitution of concepts," and that it is by their "full maturity, and not in the process of their constitution, that concepts and functions necessarily intersect, each being created only by their specific means" (1994, p. 161). They mean, quite literally, that a philosopher's concept does not replace the scientist's function, and vice versa. Rather, their argument places the foci upon intersections where concepts cohere with functions, such as the ethical dimensions of using artificial intelligence and machine learning techniques for distributing bank loans or managing health care intersecting with issues of racial and gender biases within the code bases and data sets used to reach decisions. Software does not require the definitional work it has inhered over its analysis within media studies; it already has one that is sufficient to its scientific and technical purposes.

Software is a fetishistic, ephemeral medium only in so far as it derives its meaning from distant services and largely unreadable processes encoded as machine instructions: it is true, for instance, that the code implementing business logics—e.g., the implementation itself—often runs

المنارة للاستشارات
www.manaraa.com

and acts behind the scenes of an opaque user interface, triggered by event data such that its machinations in some far away data center can never be fully known or observed. These issues are further complicated when we look at automated tasks, at processes that are instigated by seemingly 'unknowable' processes under exceptional circumstances that apparently are noticed only when we intersect with them in one's daily life. On the face of it—or at its most ghostly—software would seem to be all of the things that media studies' scholars like Chun and Kittler (1999, 2008, 2010, 2013) claim it is: arbitrary electrical signals at its most granular level, the toggling of transistor switches billions (or trillions) of times a second, which is unknowable because of its arbitrariness, because electricity is a tautology in its simplest form. But clearly software is something more than electricity (and that it does in fact exist in a less than ghostly, ephemeral state), because it emerges from aggregates of human and technical relations, becoming something more than mere signals passing as electrons through the transistors comprising the machine instructions inside of the various computational processing units that cause it to happen. The move to overly reduce it, e.g., "There Is No Software" in the Kitterlian sense, does not increase knowledge of how it is made, why it is made, how it works, what its limitations are, and so forth. Nor do the 'processes' of software seem to encapsulate their own *dasein*, or experience of becoming, to borrow a Heidegerrian concept for a moment because it relates to a narrative of 'recursion' within Kittlerian media studies', which have endowed them with an agency that allows them to increment and tick its version numbers ever upward; software is—for the time being and in a historical sense—the result of a supremely human effort relying as much on social factors as technical ones.

Software, as this dissertation has shown—for the time being and for the immediate organically human-bound future, at least—does not design itself. The practices and patterns

المنارة للاستشارات
www.manaraa.com

embroiled in its production constitute an iterative and axiomatically bounded sociotechnical regime brought forth by the problems of solving problems computationally. Software engineering is knowable, and if software engineering is knowable, it follows that software can be understood for what it is, a distinctly collaborative human and technical relation transducing signifying and asignifying practices, rather than abstracted across space and time as ghostly vestiges of one sort or another. So, what appears to be recursive in software, i.e., the manifold, ghostly, ubiquitous, and self-referencing 'being' of software, leads to an idea that it produces itself which is reinforced by expositions about 'automated programming,' such as the one Chun (2011), but ignores the generative agency of the human-technical relationship. Automated programming is not design, and can only simplistically be thought of as 'computers writing code for computers'; it is more accurate to state that automated programming is 'computers transducing diagrammatics' in the Guattarian sense: automated programming 'automates' the translation of operands into machine instructions, translating an assembler operation like 'MOV EAX, [EBX] * 10h' into byte-code for an Intel x86 processor, from a diagrammatic source, like source code, which itself is a product of design which follows from an intent to perform a specific task for a specific purpose. It is a disservice to the concept of software itself to define it in lieu of adequate knowledge of software engineering: software is difficult to design and implement, and it is difficult to design and implement precisely because it intersects with and is embroiled within sociotechnical relations. Understanding it as the product of a sociotechnical regime de-fetishizes software, making it, if not 'plain,' at least concrete, a concept firmly connected to the practices producing it.

المنارة للاستشارات

www.manaraa.com

Contributions

This dissertation primarily contributes to future media studies scholarship by offering a basis for connecting theory to practice in a Deleuze and Guattarian way. Understanding software through both Deleuze and Guattarian concepts and industry and Computer Science definitions facilitates the bridging of different communities of practice. Thus, this dissertation does not provide a new definition for 'software,' but rather models a way of understanding, from an epistemological and ontological perspective, how Deleuze and Guattari work can elucidate and operate both in media studies and in human and technical domains. Ultimately this project hast shown that software engineering is a way of attempting to provide the best computational solution for a problem; its axiomatics—its formalized practices—function best when they evaluate a problem with its conditions. This recognition fundamentally ties engineering practices to a kind of problematization envisioned by Deleuze (1994) in *Difference and Repetition*, which resounded throughout his collaboration with Guattari in *What is Philosophy* (1994)*?* Scientific practices are oriented around a plane of reference, which seeks to organize and describe relations in a state of affairs in terms of functions and propositions. Functions work by changing the state of something, by slowing down time such that relations can be individualized, enumerated, described, and enacted in predictable, and reproducible ways. At its purest, a plane of reference attempts to form the basis for a set of abstractions that describe how something works, leading to functions that modify or impact the way something is made to work. It can have meaning that is distinct to itself, or that is generalized across sign-points (in the Guattarian sense) such that functions or axioms have relations to materializations, e.g., chemical reactions, or the model of an atomic particle. But planes of reference cannot always consist of abstractions when the types of work they serve emphasize social relations: wile engineering practices rely on the abstractions

المنارة للاستشارات
www.manaraa.com

produced by 'purer' sciences, they encode, functionalize, and axiomatize functions used for translating, understanding, and solving problems, like proposal writing, needs assessments, requirements gatherings, documentation practices, and in the case of software engineering, communicative tools and practices amongst developers within a specific Software Development Life Cycle (SDLC), like Agile. At some point 'needs'—originating from a social milieu—must be communicated to those performing the labor associated with developing software. Software engineering encapsulates a plane of reference that resides in constant tension with those constituted by Computer Science and managerial praxis and has developed its own ways of negotiating problematizations to produce software in response to problems arising from a social milieu. Adequate definitions of 'software' require that 'software engineering' be understood and treated self-sufficiently, rather than over-coded and discarded in favor of a fetishistic narrative.

The major contribution of this study is a framework thinking through concepts from the perspective of their problematics, which rhizomatically connects to many human and technical relations. Some relations are plain to see, whilst others are hidden or discarded by normalized historical accounts, in the Kuhnian sense. The problem-oriented sociotechnical regime provides a basis for understanding the other side of a social, human, and technical equation that tends to be dominated by philosophical concepts. By mobilizing Deleuze and Guattarian arguments and concepts, the framework allows industrial and scientific definitions of functions and propositions to be integrated—by way of intersection—into current media studies scholarship. Such intersection is premised on the self-sufficiency and independence of the work the thought-forms Deleuze and Guattari defined. Thus, the development of the concept of the sociotechnical regime and its emphasis on transduction as a process within the problematization of software is an effort to adequately define software for media studies in a way that emphasizes intersectionality and

المنارة للاستشارات
www.manaraa.com

collaboration. Mapping out the framework, exploring the early history of computation and the formalization of software engineering, and defining transduction as a process which straddles signifying and asignifying boundaries, shows how software is a constant negotiation of social and technical relations. Software is a kind of media of the problematic and is the product of relations between things that do not always lend themselves to clarity. The task of producing software therefore is a negotiation of human factors playing out across a technical plane of reference: how well a product's purpose is communicated determines the extent to which it is understood; social, signifying practices can interfere with otherwise asignifyingly technical ones.

Software does not emerge into the world a tremendous amount of human effort, and 'meaning' to software is relative to itself, while being culturally and societally bound in human spheres. Software is one kind of culminating product of social and machinic relations, and to think in terms of relations is Deleuze and Guattarian, such that what comes about as software *becomes* through sets of interactions and axiomatics within a state of affairs, between human and nonhuman actors faced with overcoming technical and communicative limitations while leveraging available affordances. Understanding software engineering as a type of sociotechnical regime, as this dissertation has argued, thus allows software itself to be a media effect of a problematic encounter between practitioners and stakeholders acting to solve it computationally. If media studies redefine software in light of software engineering, software is no longer fetishistic, but rather the product of processes and practices that begin and culminate through communicative and social acts. Software is always visible and is evidence of an underlying problem and its conditions; the framework of the sociotechnical regime is distinctly suited to mapping the human and technical intersections producing it.

المنارة للاستشارات
www.manaraa.com

Summaries

This dissertation made a sustained argument about software engineering in three parts: it outlined a framework for problem-oriented sociotechnical regimes using Deleuze and Guattarian concepts; it examined the history of early computation and the formalization of 'software engineering' in 1968, declaring it to be its own sociotechnical regime distinct from Computer Science and managerial praxis; and it defined the concept of transduction, based on Guattari's mixed semiotics, to describe the intersectionality of how signifiers and asignifiers are converted across domains, e.g., how 'talking about' software in signifying terms is translated into the asignifying diagrammatics of source code and computer processes. The project develops an understanding of 'software' that can no longer be fetishized, and is no longer ghostly, and connects the practices of software engineering to a concept of software that is rooted in the self-sufficient definitions of the disciplines that instantiated them.

Chapter 2 of this dissertation forwarded the concept of the problem-oriented sociotechnical regime. It set the basis for this by examining crucial concepts from Deleuze and Guattari, developing the framework for understanding how problems organize and coalesce into practices and identities, e.g., software engineering and individually, a software engineer. Sociotechnical regimes are relatively stable sets of technical and social practices, recursive values, identities, and productive outcomes that coalesce around one or more problematic encounters (which are the combination of a problem with its conditions). Computer science, for example, seeks to solve problems computationally, in a generalizable sense, and has standardized on the algorithm as a unit of work and measure and testability for the efficacy of their solutions. The chapter began by examining and defining Deleuzian problematics; it explored problem-orientations and their role in a regime's creation of its collective identity; and it explained, in

المنارة للاستشارات
www.manaraa.com

granular detail, how Deleuze and Guattari's concepts contribute to the notion of a rhizomatic binding of human and technical relations organized to produce solutions. The first part set up the discussion of the history of software engineering from the perspective of a sociotechnical regime, such that the problems of software engineering could be appreciated as related, but distinct from, those of Computer Science and managerial praxis.

Chapter 3 examined the history of early software engineering, exposing along the way the kinds of problematics it was organized to solve. It interpreted 'software engineering' as a sociotechnical regime, discovering its distinctness from Computer Science and managerial praxis, and highlighting the tension that exists between it and its adjacent fields. The chapter demonstrated how Deleuze and Guattarian concepts worked their way through the organization of a discipline that examined its processual development over time. Understanding software engineering as a sociotechnical regime demonstrates how to think through software problematically, because of human and technical relations in negotiation with a broader social impetus. The chapter places the onus of analysis less on programmatic statements and source code or the idea of the 'programmer' as an author and more on the technical factors mediated by social practices leading to their actualization as software, effectively de-fetishizing software. The framework of the sociotechnical regime described in Chapter 3 allows definitions of 'software' to re-integrate industry and Computer Science meanings, providing a future basis for expanding the intersectionality of media studies' scholarship investigating the principal technology at work in our common state of affairs. The key to understanding 'software' as the productive effort of a regime is a consideration of problems being immanent to their conditions, which influence those conditions as much as they are shaped by them. Understanding the problematics that lead to the rise of software engineering produces the means to detail the conditions which determine how

للاستشارات المنارة
www.manaraa.com

the becoming of a thing happens. Deleuze and Guattari's rhizomatic thinking across intersecting plateaus and events and their emphasis on processual, continual *becoming* is ideal for analyzing and understanding how software is made and what it is once it begins to perform work in a state of affairs. The history of software engineering and the analysis of how software is produced concretize and de-fetishize its nature, deflating 'programmer' from 'software engineering' by showing implementation to be one phase and one concern among many. Software is not only the culmination of distinctly technical expertise, but of communicative expertise within practices of design, maintenance, testing, implementation, and of labor management.

Finally, Chapter 4 examined the communicative aspects of how a regime and its actors interact with the world through Guattarian mixed-semiotic processes of transduction. The chapter defined Guattari's concept of asignification and its role in the diagrammatism of software engineering processes, locating them in the plane of reference that organizes the limiting fields and assemblages of enunciation which define its sociotechnical regime. Next, it defined the concept of transduction as a process mediating the actual movement and conversion of values and their meanings into and out of asignifying and signifying domains. Finally, it explained signification's role in transducive processes by specifically examining the difficulties inherent to the double articulation of a problematic. Transduction and mixed semiotics account for the problem of communicating problems, which is evident in the historic unreliability of software and the unpredictability of its implementation and delivery. Software has historically been unreliable. It is a difficult task to translate needs and processes described in terms of signification into asignifying diagrammatics. Chapter 4 explained what is gained and lost when a process is translated across domains, while detailing some of the axiomatics (like 'best practices') that have

المنارة للاستشارات
www.manaraa.com

been formalized within software engineering to account for the difficulties inherent to turning signifiers into asignifiers, and vice versa.

Fully considered, the framework of the problem-oriented sociotechnical regime, the examination of software engineering as a regime, and the development of the idea of transduction as a process at the point of intersectionality between practitioners of a regime and a socius or state of affairs develops an idea of 'software' for media studies that is nuanced and sophisticated. Software is not ephemeral if it is connected to the practices that produce it; like Siegert (2018) said, concepts should connect to practices. Deleuze and Guattari's concepts, and Deleuze's focus on problematization, provide a way to reconnect an operational definition of 'software' to the communicative practices producing it. Once performed, software can no longer be fetishized, programmers and source code can no longer be taken as emblematic for 'software engineering,' and the materiality of software becomes rhizomatic, but explicitly concrete. Software is the conjunction of human and technical relations and is fundamentally a diagrammatic projection of a solution—a computational model—in response to a problem that is designed to account for the problem's conditions.

Applications

There are three major areas the framework of the problem-oriented sociotechnical regime applies to in media studies work that can affect future discussions of software's materiality and status as a potential form of media:

(1) by focusing scholarly attention on the problematics that give rise to software, the framework broadens discussions of the technology to include the many human and technical relationships required to produce it, such that software—which is always

المنارة للاستشارات
www.manaraa.com

designed into existence—becomes a 'solution' to an issue by cohering around a problematic, and is implemented using processes that are *designed* to respond, systematically, to that problematic and its rhizomatic human and technical relations;

(2) by incorporating Deleuze and Guattarian thinking, such as their argument for the self-sufficiency of concepts and functions, the work of mapping software engineering as a sociotechnical regime discovers and premises points of intersectionality as the basis from which to create new concepts, while being free from needing to redefine the functions inherent to that regime, making the framework inherently interdisciplinary, thereby allowing it to connect concepts to practices;

(3) the media effects of software can be analyzed through a lens of mixed semiotics focusing on the processes which produce it, for many of its consequences can be enumerated if its planning, design, testing, and maintenance are explored as transducive interactions that shed and inhere significations and asignifications to produce a diagrammatic model that attempts to solve a problem.

At its core, a sociotechnical regime is organized in response to a problematic; software engineering, for instance, required the problem of communicating problems computationally before it could be formalized as a discipline with its own sets of patterns and practices. In Computer Science's case, the development of 'computers' provided the tools necessary to solve problems computationally; the issue that software engineering evolved to tackle, then, became more about how problems can be reliably understood to such an extent that they can be translated into the forms that computers operate on. This translation, through tasks like requirements gathering and design work, is implemented as software, which is a form of solution to a problem fed to a regime dedicated to its production. The problem-oriented sociotechnical regime models a

المنارة للاستشارات
www.manaraa.com

Deleuze and Guattarian approach to what one might call a medium of problematics, of software that coheres around processes of mediation, communication, and implementation. The software is the practice, while the problematic is the concept.

Future Directions

This project had to avoid delving into broader cultural and gender-related issues to limit its scope. There is a tremendous amount of work that can and should be done to connect the practices of software engineering, computer science, and managerial practices to issues of cultural and gender-based theories and histories. Wendy Chun detailed much of what early women programmers ("coders") experienced, but I would like to engage at a broader historical and systemic level to discover the ways in which the axiomatics of a sociotechnical regime might perpetuate biases and status quos: if the plateaus across which a regime stretches are rhizomatic, agents within them can conceivably work to limit the lines of flight that would allow those rhizomes to encapsulate broader cultural knowledges and values. Ideally part of the future work related to this topic would produce a rich historical account and systematic analysis of software engineering to discover additions and truncations of cultural and gender-related issues.

The work of mapping software engineering as a sociotechnical regime emphasizes an area of Deleuze and Guattarian scholarship that has thus far been underdeveloped. Deleuze and Guattari have provided a philosophical basis for connecting practices to concepts, which can lead to productive interdisciplinary efforts for studying the media effects of machine learning systems, artificial intelligences, and the pervasive software systems operating in our collective state of affairs. Software, rather than being ghostly, is supremely knowable if the practices used to produce it should exist as they are defined within their respective thought-forms, without

المنارة للاستشارات
www.manaraa.com

overcoding them through esoteric definitions of materiality or conflations of programming and writing or source code as text. This respect allows software to be realized and materialized as a problematic encounter, a point of intersection between a set of practices that plan for it, design it, implement it, test it, and maintain it and the state of affairs which necessitated its existence. There are several directions this work can grow: first, problematics can be broadly expanded upon as a basis for connecting concepts and practices; second, a discussion of the media effects of a problem-oriented sociotechnical regime can be further developed, so that software engineering (or other disciplines) can be described as mediating a problematic encounter; and a method for broadly interpreting communication at points of intersection using mixed semiotics can be granularly implemented by using a case study approach to existing software projects using tools like git to recover data and mixed methods for examining the efficacy of efforts to understand and translate requirements into designs and implementations that act as solutions. Understanding software from the perspective of software engineering de-fetishizes it, making its problematic encounter—i.e., its reason for existing—far more important than attempting to wrangle through issues of its materiality.

المنارة للاستشارات
www.manaraa.com

REFERENCES

Alpaydin, E. (2016). *Machine learning: the new AI*. Cambridge, MA: MIT Press.

Ambler, S. W. (n.d.).

Amott, S., & DePaul University. (1999). In the Shadow of Chaos: Deleuze and Guattari on Philosophy, Science, and Art. *Philosophy Today*, *43*(1), 49–56. https://doi.org/10.5840/philtoday199943136

Apple. (n.d.) Core Motion. Retrieved from https://developer.apple.com/documentation/coremotion

Atchison, W. F., Schweppe, E. J., Viavant, W., Young, D. M., Conte, S. D., Hamblen, J. W., … Rheinboldt, W. C. (1968). Curriculum 68: Recommendations for academic programs in computer science: a report of the ACM curriculum committee on computer science. *Communications of the ACM*, *11*(3), 151–197. https://doi.org/10.1145/362929.362976

Bergson, H. (1988). *Matter and memory*. New York: Zone Books.

Boehm, B., & Basili, V. R. (2001). Top 10 list [software development]. *Computer*, *34*(1), 135–137. https://doi.org/10.1109/2.962984

Brooks, F. P. (1995). *The mythical man-month: essays on software engineering* (Anniversary ed). Reading, Mass: Addison-Wesley Pub. Co. Retrieved from http://www.csee.umbc.edu/~mgrass2/cmsc345/paper%20-%20MythicalManMonth.pdf

Campbell-Kelly, M., Aspray, W., Ensmenger, N., Yost, J. R., & Aspray, W. (2014). *Computer: a history of the information machine* (Third edition). Boulder, CO: Westview Press, A Member of the Perseus Books Group.

Ceruzzi, P. E. (2012). *Computing: a concise history*. Cambridge, Mass: MIT Press.

المنارة للاستشارات
www.manaraa.com

Chun, W. H. K. (2008). On "Sourcery," or Code as Fetish. *Configurations*, *16*(3), 299–324. https://doi.org/10.1353/con.0.0064

Chun, W. H. K. (2011). *Programmed Visions: Software and Memory*. The MIT Press.

Chun, W. H. K. (2016). *Updating to remain the same: habitual new media*. Cambridge, MA: The MIT Press.

Colebrook, C. (2002). *Understanding Deleuze*. Crows Nest, N.S.W: Allen & Unwin.

Conway, B., J. Gibbons, and D. E. Watts. *Business Experience with Electronic Computers: A Synthesis of What Has Been Learned from Electronic Data Processing Installations*. New York: Price Waterhouse, 1959.

Bureau of Labor Statistics. (2019). *Computer and Information Technology Occupations*. Retrieved from https://www.bls.gov/ooh/computer-and-information-technology/home.htm

Day, R. W. (2017). *Design error: a human factors approach*. Boca Raton: CRC Press, Taylor & Francis Group.

De Landa, M. (2010). *Deleuze: history and science*. New York: Atropos.

Deitel, H. M., & Deitel, P. J. (1994). *C: how to program* (2nd ed). Englewood Cliffs, N.J: Prentice Hall.

Deleuze, G. (1990). *The logic of sense*. New York: Columbia University Press.

Deleuze, G. (1994). *Difference and Repeition*. New York: Columbia University Press.

Deleuze, G. (1995). *Negotiations, 1972-1990*. New York: Columbia University Press.

Deleuze, G., & Guattari, F. (1977). *Anti-Oedipus: capitalism and schizophrenia*. New York: Viking Press.

المنارة للاستشارات

www.manaraa.com

Deleuze, G., & Guattari, F. (1987). *A thousand plateaus*. Minneapolis: University of Minnesota Press.

Deleuze, G., Guattari, F., Tomlinson, H., Burchell, G., & Rogers D. Spotswood Collection. (1994). *What is philosophy?* New York: Columbia University Press.

Department of Defense. (1985, June 4). Military Standard: Defense System Software Development. Department of Defense. Retrieved from http://everyspec.com/DoD/DoD-STD/DOD-STD-2167_278/

Developer. (2019). In *PC Magazine's Encyclopedia*. Retrieved from https://www.pcmag.com/encyclopedia/term/41187/developer

Dijkstra, E. W. (1988). On the cruelty of really teaching computing science. Retrieved from https://www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html

Dinnen, Z. (2013). 'Break out that Perl script': The imaging and imagining of code in The Social Network and Catfish. *European Journal of American Culture*, *32*(2), 173–186. https://doi.org/10.1386/ejac.32.2.173_1

Ensmenger, N. (2010). *The computer boys take over: computers, programmers, and the politics of technical expertise*. Cambridge, Mass: MIT Press.

Everett, G. D., & McLeod, R. (2007). *Software testing: testing across the entire software development life cycle*. [Piscataway, NJ] : Hoboken, N.J: IEEE Press ; Wiley-Interscience.

Foucault, M. (1982). *The archaeology of knowledge*. New York, NY: Pantheon Books.

Foucault, M. (1995). *Discipline and punish: the birth of the prison*. (A. Sheridan, Trans.) (2nd Vintage Books ed). New York: Vintage Books.

www.manaraa.com

Foucault, M., Bertani, M., Fontana, A., Ewald, F., & Macey, D. (2003). *Society must be defended: lectures at the Collège de France, 1975-76* (1st ed). New York: Picador.

Foucault, M., & Gordon, C. (1980). *Power/knowledge: selected interviews and other writings, 1972-1977* (1st American ed). New York: Pantheon Books.

Galloway, A. R. (2012). *The interface effect*. Cambridge, UK ; Malden, MA: Polity.

Genosko, G. (2014). *Information and Asignification* (p.). FOOTPRINT. https://doi.org/10.7480/footprint.8.1.798

Glass, R. L. (Ed.). (1998). *In the beginning: personal recollections of software pioneers*. Los Alamitos, Calif: IEEE Computer Society Press.

Guattari, F. (1984). *Molecular revolution: psychiatry and politics*. Harmondsworth, Middlesex, England ; New York, N.Y., U.S.A: Penguin.

Guattari, F. (1995). *Chaosmosis: an ethico-aesthetic paradigm*. Bloomington: Indiana University Press.

Guattari, F. (2011). *The machinic unconscious: essays in schizoanalysis*. Los Angeles, CA : Cambridge, Mass: Semiotext(e) ; Distributed by the MIT Press.

Hauptmann, D., & Radman, A. (2014). *Asignifying Semiotics as Proto-Theory of Singularity: Drawing is Not Writing and Architecture does Not Speak* (p.). Delft University of Technology. https://doi.org/10.7480/footprint.1.794

Hayles, K. (1999). *How we became posthuman: virtual bodies in cybernetics, literature, and informatics*. Chicago, Ill: University of Chicago Press.

Hayles, K. (2005). *My mother was a computer: digital subjects and literary texts*. Chicago: University of Chicago Press.

المنارة للاستشارات
www.manaraa.com

Jason. (2016). How to tell if the iPhone is face down or face up. *Stackoverflow.com*. Retrieved from https://stackoverflow.com/questions/36097204/how-to-tell-if-the-iphone-is-face-down-or-face-up

Jones, C. (2014). *The technical and social history of software engineering*. Upper Saddle River, NJ: Addison-Wesley.

Kelleher, J. D., & Tierney, B. (2018). *Data science*. Cambridge, Massachusetts: The MIT Press.

Kitchin, R., & Dodge, M. (2011). *Code/space: software and everyday life*. Cambridge, Mass: MIT Press.

Kittler, F. (2008). Code (or, How You Can Write Something Differently). In M. Fuller (Ed.), *Software Studies* (pp. 40–46). The MIT Press. Retrieved from http://mitpress.universitypressscholarship.com/view/10.7551/mitpress/9780262062749.001.0001/upso-9780262062749-chapter-6

Kittler, F. A. (1990). *Discourse networks 1800/1900*. Stanford, Calif: Stanford University Press.

Kittler, F. A. (1999). *Gramophone, film, typewriter*. Stanford, Calif: Stanford University Press.

Kittler, F. A. (2010). *Optical media: Berlin lectures 1999* (English ed). Cambridge, UK ; Malden, MA: Polity.

Kittler, F. A., & Gumbrecht, H. U. (2013). *The truth of the technological world: essays on the genealogy of presence*. Stanford, California: Stanford University Press.

Kohanski, D. (2000). *Moths in the machine: the power and perils of programming*. New York: St. Martin's Griffin.

Koopman, C. (2013). *Genealogy as critique: Foucault and the problems of modernity*. Bloomington: Indiana University Press.

المنارة للاستشارات
www.manaraa.com

Kraemer, F., Overveld, K., & Peterson, M. (2010). Is there an ethics of algorithms? *Ethics and Information Technology*, *13*(3), 251–260. https://doi.org/10.1007/s10676-010-9233-7

Kuhn, T. S. (1996). *The structure of scientific revolutions* (3rd ed). Chicago, IL: University of Chicago Press.

Linger, R. C., Mills, H. D., & Witt, B. I. (1979). *Structured programming, theory and practice*. Reading, Mass: Addison-Wesley.

Liu, Trista (2017). "6 Essential Tips on How to Become a Full Stack Developer." Retrieved from https://hackernoon.com/6-essential-tips-on-how-to-become-a-full-stack-developer-1d10965aaead

Mahoney, M. S. (2008). What Makes the History of Software Hard. *IEEE Annals of the History of Computing*, *30*(3), 8–18. https://doi.org/10.1109/MAHC.2008.55

Manovich, L. (2013a). *Software takes command: extending the language of new media*. New York ; London: Bloomsbury.

Manovich, L. (2013b). The Algorithms of Our Lives. *The Chronicle of Higher Education*. Retrieved from https://chronicle.com/article/The-Algorithms-of-Our-Lives-/143557/

Massumi, B. (2002). *Parables for the virtual: movement, affect, sensation*. Durham, NC: Duke University Press.

May, T. (2005). *Gilles Deleuze: an introduction*. New York: Cambridge University Press.

McCracken, D. D. (1961). The Human Side of Computing. *Datamation*, *7*(1), 9–11.

Mendelson, E. (1990). Second Thoughts about Church's Thesis and Mathematical Proofs. *The Journal of Philosophy*, *87*(5), 225. https://doi.org/10.2307/2026831

Meyer, B. (2013, April 4). The origin of "software engineering." Retrieved February 26, 2018, from https://bertrandmeyer.com/2013/04/04/the-origin-of-software-engineering/

المنارة للاستشارات

www.manaraa.com

Mitchell, W. J. T., & Hansen, M. B. N. (Eds.). (2010). *Critical terms for media studies*. Chicago ; London: The University of Chicago Press.

Naur, P., & Randell, B. (Eds.). (1969). *Software Engineering: Report of a conference*. NATO. Retrieved from http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF

Niquette, P. (1995). "Introduction: The Software Age." Retrieved December 16, 2019, from http://www.niquette.com/books/softword/part0.htm

Olszewski, A., Woleński, J., & Janusz, R. (Eds.). (2006). *Church's thesis after 70 years*. Frankfurt: Ontos-Verl.

Osborne, T. (2003). What is a Problem? *History of the Human Sciences*, *16*(4), 1–17. https://doi.org/10.1177/0952695103164001

Ottino, J. M. (2013). View from the Intersection: Why We Need the Humanities and Arts. Retrieved October 21, 2018, from https://www.mccormick.northwestern.edu/magazine/fall-2013/view-from-the-intersection.html

Patrick et al. (1962). The RAND Symposium: 1962; Part One - On Programming Languages. *Datamation*, (62), 25–32.

Price Waterhouse. (2018). "PwC and you." *Pwc.com*. Retrieved from https://www.pwc.com/us/en/about-us.html

Problem. (2019). In *Oxford Dictionaries*. Retrieved from https://www.bing.com/search?q=define+problem&pc=MOZI&form=MOZLBR

Process. (2019). In *Oxford Dictionaries*. Retrieved from https://www.bing.com/search?q=define+process&pc=MOZI&form=MOZLBR

المنارة للاستشارات

www.manaraa.com

Programmer. (2019). In *Oxford Dictionaries*. Retrieved from https://www.bing.com/search?q=programmer&pc=MOZI&form=MOZLBR

Rahul. (n.d.). “What is Waterfall Model In Software Engineering?” *Techno Trice*. Retrieved from https://www.technotrice.com/what-is-waterfall-model-software-engineering/

Randell, B. (1994). The origins of computer programming. *IEEE Annals of the History of Computing*, *16*(4), 6–14. https://doi.org/10.1109/85.329752

Riordan, M. (2007, December 1). The Silicon Dioxide Solution: How physicist Jean Hoerni built the bridge from the transistor to the integrated circuit. Retrieved September 21, 2018, from https://spectrum.ieee.org/tech-history/silicon-revolution/the-silicon-dioxide-solution

Sample, M. L. (2013). Criminal Code: Procedural Logic and Rhetorical Excess in Videogames, *7*(1). Retrieved from http://digitalhumanities.org/dhq/vol/7/1/000153/000153.html

Software. (2019). In *Dictionary.com*. Retrieved from https://www.dictionary.com/browse/software

Software. (2019). In *Oxford Dictionaries*. Retrieved from https://www.bing.com/search?q=definition+software

Smith, D., & Protevi, J. (2015). Gilles Deleuze. In E. N. Zalta (Ed.), *The Stanford Encyclopedia of Philosophy* (Winter 2015). Metaphysics Research Lab, Stanford University. Retrieved from https://plato.stanford.edu/archives/win2015/entries/deleuze/

Stengers, I. (2005). Deleuze and Guattari’s Last Enigmatic Message. *Angelaki*, *10*(2), 151–167. https://doi.org/10.1080/09697250500417399

The Next Web. (2016). Why half of developers don’t have a computer science degree. Retrieved from https://thenextweb.com/insider/2016/04/23/dont-need-go-college-anymore-programmer/

المنارة للاستشارات

www.manaraa.com

Transducer. (2019). In *Oxford Dictionaries*. Retrieved from https://www.bing.com/search?q=define+transduce&form=OPRTSD&pc=OPER

Turing, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem, 230–265.

Ware, W. H. (1965). As I See It: A Guest Editorial. *Datamation*, *11*(5), 27–28.

Watson, J. (2009). *Guattari's diagrammatic thought: writing between Lacan and Deleuze*. London ; New York: Continuum.

Wing, J. M. (2006). Computational thinking. *Commun. ACM*, *49*(3), 33–35. https://doi.org/10.1145/1118178.1118215

Winthrop-Young, G. (2011). *Kittler and the media*. Cambridge: Polity.

المنارة للاستشارات

www.manaraa.com

Reproduced with permission of copyright owner. Further reproduction prohibited without permission.