**Computational Culture**  
a journal of software studies

**Spaces of Flaws of Flows: COBOL and the back-back-ends of development**

**ARTICLE INFORMATION**

- **Author(s):** Linda Hilfling Ritasdatter  
- **Affiliation(s):** School of Arts and Communication, Malmö University  
- **Publication Date:** July 2023  
- **Issue:** 9  

**ABSTRACT**

Software is often described according to a front-end and back-end dichotomy, with interaction happening in the former and the latter being the domain of development and maintenance. My in-depth research on and through the supposedly obsolete and highly detested programming language COBOL, goes beyond this dichotomy and reveals a back-back-end of software, as a site that makes dichotomies such as front-end and back-end possible in the first place. The persistence of COBOL as a legacy system within the digital services of the Global North is shown to be widespread, all the while these systems are maintained by workforces located in the Global South. With a focus on India, the article discusses the paradoxical situation of supposedly developed countries being dependent on labour in emerging and developing countries, in order for infrastructures to keep seeming developed. This involves processes of outsourcing and software wrapping that are described in the article along with the educational structures that are set in place for the maintenance to take place. What Manuel Castells’ once famously dubbed the space of flows of the global network society, is here discussed as having produced a ‘space of flaws’ in which a hidden workforce makes sure that the global flows keep flowing. Beyond the context of software development, I have engaged with COBOL intersectionally, through fieldwork and practice-based art and design methodologies, mapping the extent of which this language, rather than being obsolete, occupies an ‘undead’ position within planetary information networks. The article suggests that the existence of such undead legacy systems goes against ideas of linear technological development as a natural force as well as linear socio-economic development on a whole. The temporal and spatial regimes that work to hide asymmetries of development are discussed and ultimately, I suggest that, contrary to this obfuscation, we need analytical and practical approaches that actively engage these asymmetries.
Introduction

Between the years of 2013 and 2020, I carried out a multi-layered, intersectional interrogation of the supposedly obsolete programming language COBOL, with the aim to provide insights into aspects of global information architectures that are otherwise hidden and inaccessible. This led me to engage with a form of human-machine interchange that, unlike the ideal of frictionless interface design, is not designed with end-users as the primary focus. On the contrary, it is a secluded form of interaction that takes place on the backside of IT systems, that sustain global flows. This, I analyze as a back-back-end of globalization, a space of flaws, containing a large group of programmers, predominantly located in India, who are constantly fixing bugs and who are maintaining the underlying information architectures of what Manuel Castells once famously dubbed the ‘space of flows’ of global networks.¹ These workers ensure a continuous and smooth seeming flow of global financial transactions, retail, ticket reservations, etc. This human labour helps to maintain and ensure that programs run as they should and that the systems of automation work, in what appears to be a perpetual and frictionless flow. As my research shows however, these are not friction-free systems, although they appear like that from the frontend user’s perspective, where one cannot see the spaces of flaws, the ongoing updating, fixing and enhancement processes of the systems and code taking place in the background in order for the smoothness to remain smooth. In this article, I will, engage with COBOL, to describe this space of flaws disguised as a space of flows, first on a material granular level, with an analysis of how a supposedly obsolete programming language continues to persist. Secondly, I will focus on the practices, labour forces and structural global inequalities that, covertly, are put in place for development to seem to keep developing, all the while Zombie legacies such as COBOL are being maintained in the hidden away back-back-end.

This maintenance work is strongly detested by the global software developer community and my analysis explores the inherent power mechanisms that manifest in such disdain. In this respect, my research could be said to follow Christian Fuchs’s work on how Western corporations exploit Indian software engineers in the name of globalization.² However, there is a further complexity to this exploitation, which exceeds the linear condition of exploitation, and for
this part of the analysis, I formulate a spatio-temporal critique of linear developmental models that link technology, space and progress together. Here, I build on feminist geographer Doreen Massey, who argued that there is a temporal dimension to any geographical distinctions and that this is a fundamental aspect of the Enlightenment and the broader concept of modernity. In the Western tradition, space became interpreted according to a universal, linear form of development in which “different ‘places’ were interpreted as different stages in a single temporal development.” Thus, the geography of the world has been transformed into a single history of the world, a history in which spatial multiplicity and heterogeneity are reduced to a universal timeline of ‘linear progress’. Nodes at various points in time indicate the divergence of distinct locations, which are identified based on the overall chronological logic using labels such as ‘backward’, ‘developing’, or ‘developed’. This, in turn, reflects how different locations are valued through their respective temporal proximity or distance from the shared, universal future. When placed in this framework, my inquiry yields significant ramifications for our comprehension of technological advancement, and our concept of development and future. Throughout the article, I discuss how the persistence of COBOL also means a dependency of the Global North on labour and knowledge in the Global South. Rather than suggesting that this form of exploitation can simply be overcome by the south ‘catching up’, I discuss a breaking with the binary of developed and developing altogether. The conclusion looks at how this could be the starting point for addressing asymmetries in the design and maintenance of information architectures, also from a practical design perspective, in ways that have consequences for future software economies and ecologies.

**What Spaces of Flows are made of (COBOL)**

On April 4, 2020, amidst the global standstill caused by the COVID-19 pandemic, New Jersey Governor Phil Murphy held a press briefing, during which he issued an urgent request for programmers proficient in what he mistakenly referred to as ‘Cobalt’. Murphy was not referring to the silvery-blue metallic chemical element Co with the atomic number 27, which is commonly used in lithium-ion batteries. His call was for programmers of the often thought to be obsolete programming language COBOL. COBOL is an acronym for COmmon Business-Oriented Language, a programming language developed by the US
Department of Defense in the late 1950s. As its name suggests, it was tailored specifically for business processing. Given its status as outdated and nearly obsolete, it is not without historical irony that Murphy would not remember its name correctly. How come COBOL surfaced in the COVID-19 crisis? What caused the shortage of COBOL programmers? And why had the language been neglected for so long?

COBOL was one of the earliest third-generation programming languages, the so-called high-level languages. In this generation, user interaction is abstracted not only from the machine, but from machine-specific languages through so-called ‘automatic programming’. As noted by Wendy Hui Kyong Chun, integral to third-generation programming languages are an initial absence, not only of the machine, but also an absence of the programmer, since the program can now be executed anywhere, anytime, independently of the programmer who initially created it. Accordingly, through third generation, high level languages, programming emerges as a supposedly indestructible and globally applicable operation. In this way, the emergence of high-level programming languages can also be said to embody an ideal of software as enabling an absolute, universal approach to space, in contrast to earlier generations of computation which were more local and site-specific.

The functionality of COBOL programs extends this universal spatial aspect even further: the language was specifically produced to spread automation in businesses by making it easier to handle electronic data management of financial transactions, logistics, inventory, payroll, ticket reservations and the like. Consequently, COBOL applications can be seen as a reinforcement of absolute space within the framework of globalization, as they facilitate the automated coordination, allocation, and transfer of resources, assets, and information.

Doomed and detested, although widely deployed

Originating in times of severe lack of programmers, COBOL was designed to be easily accessible, even to non-programmers. The people-oriented user interface strategy of renowned computer scientist and mathematician Grace Hopper, which utilized common English language to make computation
accessible to a wider user group, had a significant impact on the design of the language. However, since its early development, COBOL faced substantial criticism, and despite the good intentions, COBOL would become one of the most widely criticized programming languages of all time. (Synonymous with {evil}.) A weak, verbose, and flabby language used by {code grinder}s to do boring mindless things on {dinosaur} mainframes. Hackers believe that all COBOL programmers are {suit}s or {code grinder}s, and no self-respecting hacker will ever admit to having learned the language. Its very name is seldom uttered without ritual expressions of disgust or horror. (Eric S. Raymond, ed., “COBOL” in The New Hacker's Dictionary, Third Edition, (Cambridge, Mass.: MIT Press, 1996), 115.]

In 1960, the chairman of CODASYL, received a white marble tombstone engraved with ‘COBOL’ in golden letters. The tombstone was sent by a committee member, as a tongue-in-cheek way to express concerns about the slow progress of COBOL’s development and symbolically ‘bury’ COBOL even before it was born. This skepticism continued to cling to the language even after its launch. According to Jean E. Sammet, who played a critical role in COBOL’s initial design team, the language has never been popular. She noted:

> It is hard to pinpoint the reasons for lack of charisma in a language, and it has very little to do with the actual use. For example, BASIC and COBOL are very widely used languages, but I doubt whether many people are personally enthusiastic about either of them.

Such ‘lack of enthusiasm’ or even discomfort with COBOL has perhaps been expressed most boldly by Dutch computer scientist Edsger Dijkstra in his notorious 1975 rant on COBOL. Here Dijkstra depicted the teaching of COBOL as a ‘criminal offense’. Dijkstra’s opposition to COBOL was rooted in a systemic view of execution having to be carried out as effectively and seamlessly as possible. In his earlier, influential letter to the editor of the ACM, Dijkstra had focused on the problem of the GOTO statement, which he felt hampered the running of a program and made it messy. Dijkstra stressed that a wise programmer should do their utmost to shorten the conceptual gap between the static program and the dynamic process, to make the
correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible. COBOL, however, does not only contain GOTO statements. Its syntax, which bears a resemblance to that of a natural language, permits the creation of extensive and complex code that can result in lengthy and convoluted programs that are difficult to comprehend due to their size. As a result, the user-friendly nature of the language that was meant to make the code easily understandable has paradoxically led to it becoming hard to comprehend.

Another important reason for the dismissal of COBOL may be attributed to the language’s problem domain and the down-to-earth, everyday nature of COBOL applications. Just a few years before COBOL was conceived, computer pioneer Howard Aiken, who was also Grace Hopper’s former employer, sarcastically remarked:

> If it should turn out that the basic logics of a machine designed for the numerical solution of differential equations coincide with the logics of a machine intended to make bills for a department store, I would regard this as the most amazing coincidence that I have ever encountered.

Nevertheless, COBOL was specifically created to meet everyday computing needs, such as generating bills in a department store, managing payrolls, organizing retail, or handling reservations. These are all services, which, instead of being productive, are of a reproductive nature, as they simply ensure that the wheels of the overall production machine are turning. The field of computer science, however, has often considered these problem domains as simplistic or non-relevant, as suggested by Aiken’s statement. An example of this can be seen in Terrence W. Pratt’s textbook, *Programming Languages* published in 1975, which indicates that COBOL is primarily designed for processing business data ‘...in which the problems are (...) relatively simple algorithms’ such as computing the payroll for a sizeable organization. However, computer scientist Ben Schneiderman has argued that despite the computer science community’s inclination to perceive it as such, it is challenging to characterize a payroll system as simple.
In his rant on COBOL, Dijkstra suggested two possible courses of action for eliminating it: ‘fight the disease or pretend that it does not exist.’ A strategy of neglect was indeed widely utilized: A decade later, in the mid-1980s, it was noted that even though COBOL was reportedly used in 60% of all computer programs worldwide, leading programming textbooks at that time did not include it in their indexes.

Although the language has been largely ignored and disregarded in computer science curricula, it has been difficult to eradicate. In my research, I encountered the example of an automated contracts-administration system that had been in use for over sixty-five years and still relies on COBOL. The system is known as the Mechanization of Contract Administration Services System (MOCAS), and currently oversees contracts within the US Department of Defense. MOCAS was launched back in 1958 and in 2015, it was celebrated by MIT Technology Review as the oldest (known) software still in use.

The system is written in COBOL, and although preparations are being made for modernization, it currently has around two million lines of code consisting of a combination of nine hundred batch processing programs written in COBOL and nine hundred interactive online programs written in the mainframe programming language MANTIS, interfacing a Cincom Supra database. MOCAS is responsible for managing various types of contracts such as fixed price, cost reimbursable, multi-year contracts, financing contracts, and fund management. The system handles around 280,000 contracts annually with a total value of roughly $2.55 trillion and in 2022 had a worldwide user base of more than 17,000 active contractors.

The example of the longevity of MOCAS may seem surprising, given the centrality of planned obsolescence to the organization of modern capitalism. In 1923, Bernard London advocated for the introduction of artificial obsolescence through structural regulations, so that objects and devices could be officially declared ‘outdated’. His aim was to fuel the weak economy and bring an end to the depression through increased consumption. A decade later, Austrian economist Joseph Schumpeter characterized instability, ephemerality, and obsolescence not as something that needed to be centralized, regulated, and planned on a governmental plane, as London had demanded, but to be
understood as integral to capitalism. Schumpeter described economic structures as forces of nature that are continuously mutating, breaking down, and restructuring themselves, eventually becoming outmoded due to processes of ‘creative destruction’, which were the result of forms of radical innovation. According to Schumpeter, it was through the innovation and development of new modes of production or means of distribution, radically altering the existing systems, which lead to the older models, no matter how solid, to eventually become obsolete. The MOCAS system, with its stately 65-years seems to exist as an anamoly within this logic of development. After all, by incorporating both the principles of planned obsolescence and creative destruction, capitalism has managed to effectively enclose technological advancement as, like Schumpeter described development itself, a ‘force of nature’. This model recognizes development as an ever-evolving ‘socio-economic survival of the fittest’. As a result, within technological development, relentless upgrades are regarded as healthy and natural improvements. This leads to a situation where new versions and updates are introduced with the unchallenged assumption that they represent linear progress. Thus, an operational system like MOCAS seems to be not just a surprising impossibility, but perhaps an error or a mistake.

Back in 2007, COBOL was ranked number one on Computerworld’s ‘The Top 10 List of Dead (or Dying) Computer Skills’. But despite suggestions that legacy COBOL systems are rare exceptions and will soon become obsolete, there is abundant evidence to the contrary. As of June 2023, sixteen years later, the language is still listed as the twenty-second most widely used programming language according to TIOBE’s index of most popular programming languages. There is an obvious contradiction within these findings.

In their student promotion material, the company Micro Focus states that seventy percent of all business transactions worldwide and 95% of all ATM swipes are still made through COBOL. The language is in widespread use in numerous industries, including banking, retail, transportation, and governmental organizations. Notably, several institutions, such as the Danish tax authority SKAT, the German State Retirement system, the Swedish Social Insurance Agency, the US Social Security Administration, and the UK’s Department for Work and Pensions (DWP), all rely on COBOL.
In 2022, a worldwide survey was conducted among approximately 1,100 developers, software architects and engineers, development managers, and IT executives from 49 countries. The survey findings suggested that there are presently 775-850 billion active lines of COBOL code being utilized globally. Moreover, the number of active lines of COBOL code are increasing. Ninety-two percent of the survey’s respondents viewed COBOL as strategic and almost half of them expected that their organization’s amount of COBOL code in use would increase in the coming year. These numbers indicates that the language is alive.

A living natural language is primarily defined by its spoken/written usage, while a constructed formal language, such as COBOL, should be able to be programmed and executed. Additionally, important factors for both types of languages include their continued development and teaching. Notwithstanding, COBOL is not commonly incorporated in the computer science curricula of universities, whether it is the Global North or South, rather, the teaching mainly takes place at on-site campuses of some of India’s largest software consultancy companies. During my own research in India, I attended some of these classes and observed that the emphasis was on maintaining existing mainframe systems, rather than developing new programs. Such coding work falls under the category of software maintenance referred to as ‘legacy maintenance’. Interestingly, the ongoing rise in the volume of COBOL code can be attributed to these maintenance procedures, such as system enhancement and upgrades, modernization, and the incorporation of new features, rather than the creation of new applications in COBOL.

Although the programming language is not entirely defunct, it cannot be characterized as entirely alive either. Instead, COBOL exists in a state of limbo, being crucial to business yet considered outmoded in many respects. It has become what can be described as a ‘living dead,’ a form of zombie technology that continues to be sustained through maintenance processes outsourced to engineers in emerging and developing countries, including India.

**Maintaining the Undead**

How does the maintenance work look like on the ground? This section zooms in...
on the modalities and sites of what has been described as a programmer’s worst nightmare.

Initially, in the field of software production, there was no distinction made between those developing new programs and those revising existing programs. Changes emerged only at the beginning of the 1970s when IBM, due to economic reasons, started to partly free their software engineers from tasks involved in support by splitting them into the two groups developers and maintainers. This resulted in a divide between the engineers who were dedicated to the creation and development of new systems and those who were instead modifying programs and applications after they have been launched. From then on, only little or marginal attention was paid to the field of maintenance within computer sciences, even though, sources claim that between fifty and seventy-five percent of total software production costs are spent on maintenance.

The specific field of maintenance that deals with the upkeep of legacy systems, is widely unpopular and detested. At one of India’s largest software consultancy corporations a COBOL instructor recounts how his trainees break down in tears upon learning that they have been assigned to the mainframe and thus the COBOL maintenance stream, and in the foreword to the textbook *Working Efficiently with Legacy Code*, Robert C. Martin writes:

> Legacy code. The phrase strikes disgust in the hearts of programmers. It conjures images of slogging through a murky swamp of tangled undergrowth with leaches beneath and stinging flies above. It conjures odors of murk, slime, stagnancy, and offal. Although our first joy of programming may have been intense, the misery of dealing with legacy code is often sufficient to extinguish that flame.

Michael Feathers’ hands-on textbook on the handling of legacy systems outlines various approaches to maintenance, including rewriting, migration, in-house maintenance, and outsourcing (as detailed in his book *Working Effectively with Legacy Code*). Of these options, migration is often the most expensive and time-consuming. This approach involves moving a legacy system to a new
hardware and/or software platform.

One example is the case of the Swedish Social Insurance Agency (Försäkringskassan), that underwent a migration process carried out by the company Sogeti. The initial IT system, which was developed back in the 1970s and consisted of mainly COBOL applications running on a Bull mainframe, underwent a 1:1 migration process from Bull to UNIX in which the Bull COBOL code was converted to Unix-COBOL.\textsuperscript{49} Subsequently, due to the efforts and expenditures related to the migration process, the agency will be continuing to run their COBOL applications for many years into the future, although within a new hardware environment. In this way, the language is being kept alive. Migration processes, however, are very complex and although the Forsäkringskassan project took more than two years,\textsuperscript{50} an average migration project may take five to ten years.\textsuperscript{51} Sometimes projects need to be put on hold or even returned to the initial state of the legacy system.\textsuperscript{52} This is a both expensive and highly risky process. As an alternative way, so called ‘wrapping’ has emerged as a simpler and more cost-effective way of dealing with these aging systems.

\textbf{Wrapping the (Un)Dead}

During my on-site research within IT companies in India, I gained insights into the fundamentals of how modern e-commerce web applications can communicate with older legacy systems. One noteworthy example was a basic system used to educate students about the key principles of interfacing an e-commerce web application with a legacy system. In this case, the User Interface layer was developed using the contemporary programming language JavaScript, which interacted with the business layer running the Customer Information Control System (CICS). The web application communicated with the server layer responsible for file storage access using VSAM, as well as the mainframe database environment, DB2. This enabled the retrieval of data from the DB2 database and the integration of the database through a modern website interface.\textsuperscript{53} This approach of handling legacy systems is known as ‘legacy modernization’ or simply ‘wrapping’ and was introduced in the latter half of the 1980s as a means to extend the life cycle of aging systems.\textsuperscript{54}, which called the subroutines of the legacy system and so hid the processes of the old...
Wrapping can be likened to an API (Application Programming Interface). A backend system is encapsulated by reducing its complexity to an interface of in and outputs, through which the way it may connect and interact with other software components is defined. This procedure allows the legacy system to remain running as a black box below a new layer of code written in a more modern language. Since wrapping does not modify the existing code but simply encapsulates the legacy system into a new layer of software, it can best be described as a form of black box re-engineering process in which the legacy interface is the only part of the system, which is addressed and analyzed, while the internals of the system are ignored. Subsequently, the process of wrapping does not modify the source code, the initial system architecture, or the initial functionality. It only modifies the software’s functionality through changed means of interfacing it.\(^5\)\(^5\), \(^5\)\(^6\)

When I asked a senior engineer at IBM to pinpoint the problematics of COBOL he replied:

\textit{I think there are about five trillion lines of code that have been written in COBOL and of that there are probably around four trillion still running—That is the problem! That is really the problem. Say four trillion lines of code, if you want to rewrite it: nobody has the patience and nobody has the money to do it... So, fundamentally the reason for the longevity of COBOL is the fact of the amount of code.} \(^5\)\(^7\)

By way of explaining, the extent of Cobol as coded infrastructure is what makes it endure. Even if there were sufficient programmers to rewrite the code, the expense of doing so would be too high. As a result, it appears that the world is now locked into using COBOL. The persistence of outdated systems like COBOL is not solely attributable to the sheer amount of code, but furthermore to the restrictions that this abundance of code places on so-called progress and development.
According to the editors of a special, ‘Legacy Systems’ issue of Internet Histories, ‘Legacy systems can be thought of as outdated technologies that impose artificial constraints on the future’.\textsuperscript{58} This is aptly demonstrated by the US Federal Government’s recent spending on tasks related to Operations and Maintenance\textsuperscript{59}: during the 2015 financial year, the US Federal Government allocated over three-quarters of its annual budget to so-called Operations and Maintenance tasks.\textsuperscript{60} Significantly, out of seven thousand IT investments made by the government in 2015, 5233 investments had spent all their reserves on O&M.\textsuperscript{61} This is equivalent to more than three times the expenditures used on DME activities—Development, Modernisation, and Enhancement processes.\textsuperscript{62} The Federal Governments’ O&M expenditure has grown continuously in recent years resulting in a $7.3 billion decline in the Federal Government budget on expenditures on DME activities between 2010 and 2017.\textsuperscript{63} As a result, the efforts to maintain legacy systems are having a significant impact on the technological development processes within large corporations and institutions, as seen in this case of the US Federal Government, steering them in a non-linear direction.

The aversion to legacy maintenance tasks must be contextualized within the previously mentioned framework of understanding technological evolution as a natural force. In this perspective, legacy maintenance is not just the process of sustaining existing systems, rather it is a practice that contradicts the dominant perspective on technological development, as an unstoppable, natural force. In this sense, legacy maintenance is perceived not only as an unpleasant obligation but also as a hindrance to progress. Ironically, within this system, ‘COBOL has a very bright future,’ as the vice president of global at one of India’s largest IT-companies noted\textsuperscript{64}:

\begin{quote}
We cannot sunset most of the legacy systems around the globe. If it is three billion lines of code, it will take 300,000 programmers fifteen to twenty years. Most of them will probably stop after a while because they cannot migrate systems which are this efficient to something which will be running on another disparate platform. Volume is growing. Lines of code are growing. Customers stay in business [...]. Transaction in millions of dollars/millions of transactions.
\end{quote}
Heart has to be there—up and tick!

Through the act of wrapping, COBOL based applications are being re-engineered and thereby closed down and black boxed in order to reduce the complexity of legacy systems and keep them running. This process transforms the legacy system into an inactive but still-existing system, a 'living dead', that prevails within but is still markedly different to the ideal of linear technological advancement. COBOL takes the form of a zombie or mighty bug in the system which eventually might cause the system of linear technological development to implode. The art historian, Lars Bang Larsen notes:

The zombie pushes a horizon of empty time ahead of it; whether that time will be messianic or apocalyptic is held in abeyance. The zombie represents the degree zero of our capacity to imagine the future. How can we look over its shoulder? What future race comes after the zombie? How do we cannibalize self-cannibalization?

As a form of zombie, COBOL has a bright hereafter in a non-future-future. Namely, the sheer amount of working COBOL code around, limits the possibility for changes such as the migration to more modern programming environments. Basically, it is often too costly and there may not even be enough programmers to do it. Due to COBOL's legibility, the programming language has been immensely widely implemented, but at the same time due to the very same legibility turning illegible, it has also been immensely unpopular and thus neglected, which has resulted in a sort of marginal position. And because of this very position it seems that it is now extremely hard, if not impossible, to stop using the language. Even if another language were used, it would also, eventually, become legacy. As a result, the enduring nature of large-scale computational infrastructures raises significant implications for the understanding of future, progress, and development, particularly with respect to ecological concerns and notions of recycling and re-use. Legacy COBOL systems suggest that the prevailing discourse of capitalist development, as an evolutionary, natural force, falls significantly short of capturing the complexity of technological systems. It appears that linear development itself, is dependent on outdated and obsolete systems, which are obscured through acts of
wrapping and blackboxing. Through the hiding of the material foundations of legacy systems, the capitalist technological system is still seemingly able to continue upgrading, developing, and evolving.

Based on my research work in India, the next section of this article, will make an intersectional interrogation of how this process of black boxing operates on a larger, even planetary scale by employing concealed and displaced programmers who manage and maintain legacy COBOL systems.

**Maintaining Development**

When I began this research, I enrolled in private COBOL classes in Bangalore. Here, my teacher provided me with a renowned COBOL textbook by the author duo Stern and Stern. The textbook was published by the American publishing house Wiley & Sons. It was an ‘international student edition’ sold at a much lower price to students in developing countries than the ‘normal’ editions sold in developed countries. I was struck by the front cover, which bore a label prohibiting the sale of the textbook outside of India, Nepal, Pakistan, and several other South Asian countries.

![Figure 1: My first COBOL textbook.](image)
Wiley & Sons’ production of this edition can be interpreted as a practice with the intention to support development, with the company seemingly promoting an empowerment strategy through knowledge production. However, as my research has shown, it is unclear what a COBOL programmer in a developing country is expected to develop into. Not even according to a linear developmental model, can one assume that COBOL education is paving a country’s road to progress. Following Massey, the modern conception of space entails a model where the future is already known according to a grand narrative of the one and only universal future. A future which unsurprisingly happens to be that of the ‘developed’ nations of the Global North—Europe and Northern America. This idea of the future means that since Europe and India will eventually share the same future, they are inherently ‘the same’, it is just that India is behind. The Indian education and labour market for COBOL disturbs this narrative in a way that mirrors how the ideal of linear technological development is troubled by the programming language at the same time as it is dependent on it. As an in-depth look at this education and labour market will show, new temporal and spatial inequalities between developing and developed is being installed here, running contrary to the supposed flatness of globalization.

Infosys is the second largest software consultancy business in India. As I visit for the first time in 2014, the company has an employee count of roughly two hundred thousand, and an annual intake of nearly eighteen thousand fresh hires. At least twelve thousand of those employees are working specifically with COBOL and thirty to forty thousand employees are working with projects related to it. The company has a corporate education center in Mysore, a 126 km train ride from Bangalore. Here, the right qualifications of newly recruited employees are ensured, among others, those working with the maintenance of legacy COBOL systems. The Infosys corporation mandates that all newly recruited staff undertake a six-month residency program that entails both instruction and residential living on the corporate campus.
The compound is vast. The main building of the campus, the Global Education Center-II, is the largest building to be built in India since independence. Although inaugurated in 2009, the style of the building is Western classicism. The entrance of the building is characterized by a line of pillars, reminiscent of the entrance of Parthenon, and carefully constructed according to Vignola’s Doric order of architecture. The rest of the campus features offices, dormitories, and impressive leisure areas: an enormous mirror-glass variation of Buckminster Fuller’s geodesic dome hosts a multiplex cinema complex offering the latest Hollywood and Bollywood blockbusters. Expansive food courts are serving a diverse range of North-Indian, Chinese, and European cuisine.
Additionally, there is a vast choice of sports facilities: swimming pools, squash, tennis courts, gyms, a yoga center, and a musical lighting fountain on the main square. The campus boasts a total of fourteen thousand rooms, which potentially makes it the largest hostel in the world. The dormitories are arranged in such a way that an aerial view (or satellite, for example on Google Maps), reveal their formation to make up the letters I-N-F-O-S-Y-S. Everything is kept perfectly clean, giving the impression that the physical environment is nearly as pristine as a meticulously crafted 3D architectural rendering. Everywhere one feels copy-pasted into an advertisement campaign for a global consumerist lifestyle: it could be anywhere and nowhere.
There is no doubt that learning COBOL in India is a privilege, given that the software industry is among the highest paying fields in the country. A twenty-one-year-old trainee receives a monthly salary of twenty thousand rupees during their training residency at the corporate campus in Mysore. This salary is increased to twenty-five to thirty thousand rupees after the initial six months. Upon gaining three to six years of working experience, the trainee’s salary will reach fifty thousand rupees, which is on par with the salary of a lawyer, doctor, or university professor.\textsuperscript{76}

Since the Indian tech industry has such a large pool of available workers, it is possible, for Indian software companies to adopt a marketing strategy for their international clients by hiring engineers who are often overqualified for their roles and thus may be considered ‘ill-suited’ to the jobs they perform in terms of their level of education.\textsuperscript{77} As a result, a form of deskilling is taking place, where the Indian software worker is unfit for the work tasks because they know too much. A former Infosys employee recounts the tedium and monotony that comes with working on COBOL and legacy software maintenance, which involves primarily making minor alterations in response to external factors such as changes in data input or shifts in government regulations etc:

\[\ldots \textit{it does not require them [re: the engineers based in India working with COBOL] to do much thinking, much organizing and all the information they have learned and studied, like software design, system requirements specifications, all this is not given a chance. They do not have a chance to do that. They just have to look at the code and understand it and make small changes to the logic and make it run.}\textsuperscript{78}\]
According to the dean of the Education Center, the number of new trainees at Infosys is determined by the Delivery unit in accordance with the corporation’s business needs. Every quarter, around three hundred new employees are trained in mainframe and COBOL. That makes around twelve hundred new COBOL programmers every year. In one exceptional quarter, they trained a thousand new employees in COBOL. A COBOL instructor explains how the Delivery and Production unit regularly provides guidance and directives to the teaching team on how to structure their training. The teaching has undergone several modifications in the decade leading up to my visit. Notably, the “final student project,” which serves as the culmination of the six-month training program, originally required students to create applications from scratch and took twenty-five days to complete. However, the teaching team was advised by the production unit to modify their training approach from application development to application maintenance, in view of the requirements of the corporation. Consequently, the teaching team devoted two months of their summer to creating a simulated application for the final project, with inputs from various industry experts such as banking and insurance professionals. This ‘dummy’ application, which closely resembles an actual COBOL application used in the industry, encompasses approximately forty thousand lines of code. Intentionally, the teaching team integrated bugs and errors that require rectification, as well as the task of implementing both minor and major enhancements to the application. As a result, trainees in the mainframe-stream of Infosys are no longer required to create anything from scratch for their final project—the final trainee project has instead become an extended exercise in maintenance and bug-fixing of legacy systems.

Seen in a broader context, the mainframe maintenance carried out by the COBOL programmers in India, is not a national issue. Due to India’s history of import substitution, IBM along with other transnational corporations (Coca-Cola, Texas Instruments) left India in 1978 following the Foreign Exchange Regulation Act, which had been put in effect during the Indira Gandhi government in 1973. The policy aimed to protect India’s foreign exchange reserves, and therefore, imposed restrictions on the activities of multinational corporations within the country. Yet, when Rajiv Gandhi took over as prime minister after the assassination of his mother, Indira Gandhi in 1984, the IT sector was deregulated, and the country began to open for foreign investment as well as...
export of the Indian IT sector. Prior to the import substitution, there were two to three hundred IBM mainframe computers in India. When IBM returned at the beginning of the 1990s, they realized that in their years of absence from the Indian market, mainframes had lost territory to the cheaper PCs. That means that the COBOL system maintenance in India today primarily relies on mainframe machines located in other countries: Around half of the work is performed on machines located in North America, with approximately 25% on machines in Europe. Thus, despite their physical presence in India, the engineers remotely operate machines situated in Europe and North America. Consequently, the maintenance job of a COBOL programmer in India may be likened to that of a janitor of the information architectures of the Global North. This unacknowledged work is effectively outsourced, hidden in India as well as from the users and clients in Europe and North America.

A. Aneesh has used the term ‘virtual migration’ to describe workforces outsourced through IT technologies. Virtual Migration of software labor is characterized by two key aspects: The first is temporal integration, where different time zones are unified in real-time through information networks. The second aspect is spatial integration in which the work site is decoupled from the work performance. Aneesh notes:

... just like traditional immigrant workers, they do cross national boundaries and directly occupy some employment space in sectors of the American economy. In short, they migrate without migration.

Massey’s analysis of modernity’s approach to space, suggests that the convergence of spatial differences into temporality results in distance being not only a matter of spatiality, but also temporality. Here, migration is not solely an event where those who previously were in the periphery arrive in the center, but also a form of time travel, where the migrants depart from the past and arrive in the present. In this way migration, according to Massey, eradicates temporal as well as spatial distance, and this, she notes, makes migration ‘an assertion of coevalness’, a form of co-existence. During the millennium problem at the end of the 1990s (i.e., the infamous millennium bug) many Indian engineers migrated temporarily to the US and other countries in order to
solve the Y2K problems on site, where their skills and expertise were needed. Spatial migration thus constituted a potential bridge between engineers in India and engineers in America. Today however, as outsourced labor is increasingly organized solely through virtual migration, any possibility for coevalness disappears, there is no eradication of spatial, but only eradication of temporal distance. Hence, the engineer in India, although momentarily united with the North American client through instantaneous communication, stays in the distance as the human extension of the machines.

Despite being one of the best paid sectors in India, the salary level of an Indian software developer amounts to less than a fifth of a software developer doing exactly the same job but based in the UK.

Even the synchronized temporality of telecommunications is not devoid of differences. Despite a universal global sense of synchronic presence, it is still daytime in India when the night hits the US. Consequently, the engineer in India is capable of fixing bugs and making adjustments and minor enhancements for American customers overnight, while American engineers and clients are sound asleep. Hence, as the American clients arrive in the office the following morning, the previous day’s problems have been solved and the project has advanced overnight, which gives a sense of an unstoppable machinic flow, executing 24/7.

As Christian Fuchs have argued, even though Indian software engineers are well educated and their work is well respected in India, they are heavily exploited in the global economy. Describing this as a manifestation of what David Harvey terms as ‘new imperialism’, Fuch writes,

\[\text{The value created by Indian software engineers to a large degree does not stay in the country and does not benefit all, but rather is appropriated and owned by Western capital, which accumulates capital by selling software based on the dispossession of the value created by Indian software engineers in such a way that high exploitation rates are given.}\]

Fuchs views the exploitation of the Indian workforce as a manifestation of
racism in the context of global capitalism. At the same time he emphasizes that it represents a tendency of unequal development in India. Fuchs cites Anthony D’Costa’s description of the Indian software industry's growth as intertwined with the uneven and combined development of Indian capitalism, resulting in winners and losers, and the advancement of certain regions, cities, and groups at the expense of others. Put differently, the software industry in India exacerbates class disparities.

Research has shown that Indian software engineers foremost come from well-educated middle or upper-class backgrounds in urban or semi-urban areas. As it is unlikely that people with lower class backgrounds and less education are able to become software engineers, it is predominantly people who already possess ‘advantageous positions’ (i.e. well-educated male, upper and middle class) who are able to benefit from the software industry. This in turn reinforces the difference between urban and rural regions as key aspect of the unequal/uneven development of India.

The IT and BMP industries accounted for 7.7% of India’s GDP in 2018. However, as Fuchs points out, only a very small percentage of the Indian labor force is employed in the sector. In 2018, it amounted to a share of 0.76% of India’s total workforce, while forty-seven percent are employed within the agricultural sector. The agricultural sector however, according to the CIA World Factbook, only accounts for 15.4% (2016 est.) of India’s GDP. Fuchs concludes that since the labor force is very small it makes a ‘modest impact’ if we understand the Indian economy based on labor force, whereas if we understand it as capital, it has a very high impact. This in turn results in an unbalanced division of capital.

According to a report from Edelweiss, India held a fifty-two percent share of the total worldwide market for global outsourcing in fiscal 2017. Infosys and Tata Consultancy Services are both ranked among the top one hundred of the world’s leading Digital Companies as well as best regarded companies. According to employees at Infosys Lab, sixty percent of Infosys’s business relies on Mainframe maintenance, and they estimate that for Tata Consultancy Services it is even more.
In a postmodern approach to space, distance is compressed by the annihilation of the temporal through IT technologies, resulting in physical space being perceived as compressed and unified in time,\(^{117}\) and thereby manifesting what Manuel Castells in the 1990s dubbed ‘space of flows’.\(^{118}\) This postmodern concept of space is frequently portrayed as ‘purely spatial’.\(^{119}\) This idea presents the world as a cohesive entity, ‘a depthless horizontality of immediate connections’.\(^{120}\)

Massey argues for an awareness of the inherent ‘stasis’ in the reticular mesh of ubiquitous presents that signifies a postmodern approach to space.\(^{121}\) This ‘claustrophobic holism’ is characterized by an omnipresent present that simplifies and overlooks the complexity and multiplicity of time.\(^{122}\) Consequently, globalization is inevitable and hailed as a ‘grand narrative’, whether it is seen as an irrevocable technological determinism or unstoppable market forces.\(^{123}\) Similar to the previous discussed ideal of technological development as a force of nature, both modern and postmodern spatial discourses ultimately view time/space as teleological. In the context of globalization, modernity’s temporal sequencing of space is reduced to a binary division of those who are connected and part of the global network’s synchronicity, versus those who have not managed to catch up (read: develop) and join the network, and thus, as it appears, at least for now, are excluded.\(^{124}\)

In fact, the entire concept of outsourcing depends on a playing field which absolutely is not leveled—it is in other words an effect of difference. This was very directly articulated by the CEO from an Indian IT company, who stated that ‘the proximity with the locals was seriously undermining the growth of the IT economy in the area’.\(^{125}\) Thus, the privileged ‘globally connected’ class is separating itself from the local disconnected class. In other words, in order to become smooth, the space of flows is elevated from the local. This division is manifested spatially as flyovers, or gated residential areas, business hubs or education campuses (like for instance the Global Education Center in Mysore) without electricity breakdowns or water shortages. Subsequently, the physical segregation, not only manifests, but continuously reenacts, class segregation.

The Global Education Center in Mysore and Infosys’s headquarters in Bangalore’s Electronic City are global spaces that share similarities with how Frantz Fanon has described the colonial city, as a solid, bright, clean and
smooth place. In contrast, he wrote about the native city, as the dirty, cramped and short of resources, whether that be food, water, light or energy—‘a crouching village,’ ‘wallowing in mire’. Nonetheless, Fanon’s observation acquires a new significance when applied to these and other similar hubs throughout India. Rather than being from a foreign country, the middle class that occupies these spaces, like COBOL programmers, seeks to separate itself from the complex and chaotic local everyday life by inhabiting friction-free spaces such as gated business hubs, campuses, or residential areas. Despite their attempt to elevate and separate themselves, the middle class in such spaces is paradoxically not entirely equalized within the supposed flatness of the global market. An intersectional approach reveals that Indian software engineers are ruthlessly exploited by the global labor market, and instead of joining the spaces of flows, they rather end up becoming part of a ‘spaces of flaws’ – as the janitors, upkeepers and maintainers that makes sure that the global flows keep flowing. This raises questions about the significance of the international student edition of my COBOL textbook. It appears that the Global Education Center in Mysore, like the textbook, perpetuates colonial structures that train engineers from the Global South not to participate in global flows but to maintain them through the upkeep of global information systems, such as legacy COBOL systems. Consequently, this maintenance work reinforces the idea of progress and the future within the already-developed world.

**Conclusion**

By investigating the technological legacy of COBOL, both as part of a granular material engagement with software infrastructures and as a site of maintenance, I have tried to show how what has previously been theorized as a space of flows, is dependent on structural inequalities of a planetary scale. I have discussed how the interconnected ideas of linear technological and societal development fall short of accounting for COBOL as an undead programming language which persists where it supposedly should be obsolete. At the same time, I have demonstrated how the dependency on these very same legacy systems of a neo-colonial capitalist order, is obscured, and hidden away.

With this analysis of COBOL as a technical infrastructure in globalization, a picture is drawn of flows that flow far less smoothly than they are otherwise
portrayed. It is not a flawless, frictionless technology, but a complex, messy and interwoven becoming, where the socio-economic conditions influenced the design of the programming language itself and where cultural connotations in turn determine how it is disseminated.

Contrary to the anticipation of modern and polished systems, legacy COBOL systems present a challenge to the prevailing narrative of capitalist evolutionary development, which presents progress as an inherent force of history. These systems unveil the intricate nature of technological systems, highlighting a complex reliance on supposedly outdated or obsolete infrastructures. They disrupt established assumptions about progress and technology, leading to a scenario where the maintenance of legacy infrastructures ultimately impacts the linear development, potentially hindering further ‘development’. I discussed the paradox of how concealed legacy systems play a vital role in upholding the framework of evolutionary development, while at the same time being contradictory to them.

In the second part of the article, I zoomed in on the neglected aspect of computational execution, which is maintenance work. COBOL does not run by itself but must constantly be adjusted and adapted to the external socio-economic changes that affect software. Here I looked specifically at maintenance work and how it has been displaced to India and elsewhere for reasons discussed in the first part. From a postmodern spatial perspective, one would say that the COBOL programmers located in India become connected to spaces of flows. However, in an intersectional analysis of this maintenance work based on the previous section’s cultural analysis of COBOL, it is clear that these workers not only become part of a global world, but rather take on the role of upkeepers and janitors of information architectures, a position which rather enforces asymmetries of global North and South, as well as internal asymmetries within India.

Subsequently, a wrenched conjunction between so-called development and underdevelopment is established. Here, the overdeveloped world has its systems maintained by displaced and invisible workers, keeping the technological paths running, and allowing for the developed to occupy an idea of, not only seamlessness, but also of a future. A process that works both to
produce frictionless space—a spatial dimension—and frictionless time—a
temporality of moving forward.

What adds to the irony of the entire case, is that the so-called emerging and
developing countries are supposed to develop by learning to maintain obsolete
systems of the so-called developed countries in order for their legacy structures
not to break down. Which again turns the whole notion of ‘developing’ and
‘developed’ on its head. Thus, linear technological development, even in its
most adaptive and modular forms, as inscribed in capitalism, proves to be a
myth in which the hidden maintenance of the back-back-ends, the spaces of
flaws, works not only to produce its apparently smooth flows, but to keep it
giving an impression of being developed at all. This process involves this
constructed Global North’s neo-colonial exploitation of Indian software
engineers in order for the developed to remain developed and the developing,
to seem to continue to be developing.

The insights gained from these sections begs for a redefinition and challenging
of notions of technological development with its hype cycles and revolutions.
The past decade has witnessed a surge in buzzwords and terms like “Fourth
Industrial Revolution,” “Second Machine Age,” and “Industry 4.0.” These
buzzwords are advocating ever-more enhanced versions of the digital network
society, envisioned as a fully automated infrastructure that is expected to
facilitate new forms of disruption and capital accumulation. These
advancements are driven by the development of systems empowered by
machine learning, particularly neural networks, which have gained popularity
under the umbrella of Artificial Intelligence (AI). By looking at this in a longer
trajectory, beginning with my analysis of the design imperatives behind a
language like COBOL, we can begin to re-trace the contours of the futuristic
AI/Automation horizon. A re-tracing which should have deep implications for the
field of software design, regarding the neglected back-back ends of these
‘futures’ that are in reality already both past and present.

Throughout the article, I have highlighted the underlying material conditions
and the hidden human labor involved in promises of frictionless futures and
seamless spaces. However, as an artist-researcher and designer, I believe these
insights demand not just a shift in awareness but also in practice. How would
processes and products be shaped if we acknowledge that software and interaction design is not just user-interaction, but also the interaction taking place behind the scenes, at the backends and back-back-ends of our systems. What would happen if we started to include and connect perspectives of execution, crisis, and maintenance? Software design, research and education preoccupied with front-ends sustain the asymmetrical organization exemplified throughout this article. Instead, I would like to advocate for a more situated approach that is sensitive to the socio-political and spatial inequalities of users, maintainers, and developers alike. The existing ontologies are unsettling in their continuous, asymmetric and neo-colonial arrangements, hence they are begging to become unsettled.

**Acknowledgements**

This article is based and expanding on the chapters ‘Maintaining Legacies’, ‘Maintaining Flows’, and ‘Maintaining Development’ from my PhD thesis, *Unwrapping COBOL: Lessons in Crisis Computing* (Ritasdatter 2020). I would like to extend my warmest thanks to my COBOL teachers: the educators, engineers, and students of COBOL in India. Their generous sharing of expertise, experiences, and stories of COBOL is what have made this research possible.

**References**


Irani, Lilly. “Justice for ‘Data Janitors’.” Public Books, January 15, 2015,


published in 1975.


**Bio**

Linda Hilfling Ritasdatter PhD., is an artist researcher exploring means of control (code, organization, language) as well as geopolitical aspects of information architectures. Her practice takes the form of interventions reflecting upon or revealing hidden gaps within digital infrastructures — the place where a system fails, and its inadequacies become visible. Linda’s methodology is based on
extensive research, material engagement and field work in, for example, labour outsourcing and automation as well as investigations of programming languages and outmoded technological apparatuses and systems. Linda is currently a postdoctoral researcher at the School of Arts and Communication at Malmö University and an external Senior Lecturer in Design for Change at the Department of Design at Linnaeus University, Sweden. For the project, \textit{Labour of Automation} (2022-2025), she was awarded a three-year international post-doc grant in artistic research from the Swedish Research Council. Starting in September 2023 she will be a visiting research fellow at the Department of Media, Communications and Cultural Studies at Goldsmiths, University of London.

\textbf{Notes}

9. Since the high level languages (at least in theory) can be moved freely across any kind of computer as long as they are equipped with a compiler for the given language


12. Grace Hopper was not directly involved in COBOL’s design and development, but she was one of the two technical consultants to the Executive Committee of the consortium developing COBOL (Jean. E. Sammet, “The real creators of Cobol,” *IEEE Software* 17, no. 2 (2000):30–32).

13. This approach was already evident in Grace Hopper’s FLOW-MATIC data-processing language.

14. During a 1980 interview, Grace Hopper, who was nearly seventy-five years old at the time, discussed her objective of introducing an English-like syntax as a programming interface. Her goal was to broaden the user base by facilitating computer operation through English, as opposed to symbols. She made this decision based on her understanding that most individuals preferred an alternative method to interact with technology, avoiding the use of symbols, and English seemed to be the most viable option (Grace Murray Hopper, “Oral History of Captain Grace Hopper,” interview by Angeline Pantages, *Naval Data Automation Command*, Maryland, December 1980, https://archive.computerhistory.org/resources/access/text/2015/06/102702026-05-01-acc.pdf (accessed on June 7, 2023).


20. Dijkstra, “Go To Statement”.


22. Howard Aiken (1956) as quoted in Martin Davis, *The Universal Computer: The Road from Leibniz to Turing* (New York: W. W. Norton & Company,


29. Initially MOCAS is likely to have been written in FLOW-O-MATIC, since COBOL was not launched until 1960 (Fleishman, “What is the Oldest Computer Program”).


42. Micro Focus, “COBOL Market Shown to be Three Times Larger”.


45. Source: interview with COBOL instructor at Infosys, April 2018.


49. Sogeti, “Från Bull till Unix genom lyckad migrering på FK,”
   https://www.sogeti.se/varfor-sogeti/bevisad-kompetens/test/fenix—forsakringskassan-it/

50. Sogeti, “Från Bull till Unix.”


52. Source: interview with employee at Infosys, October 2014.

53. Source: as demonstrated during interview with COBOL instructor at Infosys,
In 1986, a system known as TGMS was implemented at the IBM Thomas J. Watson Research Center. TGMS consisted of a legacy system written in PL/1, which at that time was already fifteen years old. The legacy system was re-engineered by literally wrapping the existing application into AML/X classes.


Note that the concept of wrapping is used all the time, also beyond legacy modernization: e.g. Quicktime is a wrapper for several codecs, HandBrake is a wrapper for ffmpeg and Microsoft’s web browser Microsoft Edge is a wrapper for Google’s Chromium engine.

Source: interview with senior engineer at IBM, Bangalore, November 2014.


Source: interview with senior engineer at one of India’s largest IT-companies, October 2014.

Source: interview with senior engineer at one of India’s largest IT-companies, October 2014.


68. Massey, *For Space*, 68.

69. Massey gives the example of Africa instead of India (Massey, *For Space*, 68.)


71. Source: interview with engineers at Infosys, October 2014.

72. Source: interview with engineers at Infosys, October 2014.

73. Source: interview with the Education and Research Principal at Infosys Global Education Center, Mysore, October 2014.


76. These salary levels are based on numbers provided during an interview with the manager of a local training institute in Bangalore, March 2018.


78. Source: interview with former Infosys employee, Bangalore October 2014.

79. Source: interview with the Education and Research Principal at Infosys Global Education Center, Mysore, October 2014.

80. Source: interview with the Education and Research Principal at Infosys Global Education Center, Mysore, October 2014.

81. Source: interview with the Education and Research Principal at Infosys Global Education Center, Mysore, October 2014.

82. Source: interview with COBOL instructor at Infosys Global Education Center, Mysore, October 2014.

83. Source: interview with COBOL instructor at Infosys Global Education Center, Mysore, October 2014.

84. Source: interview with COBOL instructor at Infosys Global Education Center,
Mysore, October 2014.

85. Source: interview with COBOL instructor at Infosys Global Education Center, Mysore, October 2014.

86. Source: interview with COBOL instructor at Infosys Global Education Center, Mysore, October 2014.

87. Except a small module in the very last part of the project dealing with major enhancement.

88. The Foreign Exchange Regulation Act mandated that foreign corporations limit their equity share to a maximum of forty percent. This restriction meant that these companies could only own up to forty percent of the shares in their respective companies, requiring a minimum of sixty percent of Indian shares. (Jatinder Singh, Global Players and the Indian Car Industry: Trade, Technology and Structural Change (New York: Routledge, 2019), 28–29.)

89. Source: interview with senior engineer at IBM, Bangalore, November 2014.

90. Source: interview with senior engineer at IBM, Bangalore, November 2014.


93. Aneesh, Virtual Migration, 69.

94. Aneesh, Virtual Migration, 69.

95. Aneesh, Virtual Migration, 2.

96. Massey, For Space, 77.

97. Massey, For Space, 70.

98. In fact, the millennium bug was directly associated with COBOL. The vast majority of active COBOL code worldwide, consisting of billions of lines, was based on a two-digit year representation system. With the arrival of the new millennium, it became necessary to update this system to a four-digit format. As a result of the aforementioned neglect of COBOL in the Global North, this led the Global North to call for help from predominantly programmers in India. This, in turn, led to the outsourcing boom in India (see Linda Hilfling Ritasdatter Unwrapping COBOL: Lessons in Crisis...

99. According to Glassdoor, the average salary for a COBOL programmer in the UK amounts to £42,719 per year. On the other hand, in India, according to Payscale, the average salary for a COBOL programmer is ₹765k per year (equivalent to £7,456) (sources: https://www.glassdoor.co.uk/Salaries cobol-programmer-salary-SRCH_KO0,16.htm, https://www.payscale.com /research/IN/Skill=COBOL/Salary accessed June 6, 2023).


101. Fuchs, Digital Labour and Karl Marx, 211.

102. Fuchs, Digital Labour and Karl Marx, 206.

103. Fuchs, Digital Labour and Karl Marx, 211.

104. Fuchs, Digital Labour and Karl Marx, 203.


110. In 2018, the total Indian labor force was 521.9 million (CIA Factbook. “INDIA.” The World Factbook, 2020, https://www.cia.gov/library/publications /the-world-factbook/geos/in.html), 3,968,000 of those were employed in the IT—BPM sector (CEIC, “India IT-BPM Industry: Number of Employees,” CEIC Data, 2020, https://www.ceicdata.com/en/india/information-technology-statistics-national-association-of-software-and-service-company-itbpm-number-of-employees/itbpm-industry-number-of-employees), this amounts to 0.76% of the total workforce. Moreover, of those 0.76%, 880,000 were employed in the domestic market. The remaining employees were engaged in export related processes (IT services and software (1,970,000) and Business Process Management (1,188,000)) (CEIC, “India IT-BPM Industry.”)
111. CIA, *The World Factbook*.  
115. In 2019, Infosys was ranked number three on Forbes list of global Top Regarded Companies as well as ranked number seventy-one on Forbes list of “Top 100 Digital Companies” (Forbes 2019b and 2019). In the same year, Tata Consultancy Services was ranked number twenty-two on Forbes list of global Top Regarded Companies and ranked number thirty-eight of “Top 100 Digital Companies” (Forbes, “Top 100 Digital Companies,” Forbes, 2019, https://www.forbes.com/top-digital-companies/ and Forbes, “The World’s Best Regarded Companies,” edited by Andrea Murphy, Forbes, September 18, 2019, https://www. forbes.com/lists/best-regarded-companies/).  
116. Source: interview with employees at Infosys Lab, Bangalore, October 2014.  
117. Massey, *For Space*, 76.  
119. Massey, *For Space*, 76.  
120. Massey, *For Space*, 76.  
121. Massey, *For Space*, 76-77.  
123. Massey, *For Space*, 82.  
124. This was of course Castells’ famous division of interactive and interacted. (Castells, *The Rise of the Network Society*, 365.  
127. What I have elsewhere proposed as ‘Crisis Computing’ (see Ritasdatter, ‘Unwrapping COBOL’).