Software as Composites: A Maze of Twisty Passages

Daniel "Drex" Drexler
Center for Science, Technology and Society at Drexel University

Abstract

A study of software, the way it materializes perspectives, and the limits of the promulgations of those perspectives.

Software as Composites: A Maze of Twisty Passages

It's Software's World, We Just Live in it

[It] matters with which ways of living and dying we cast our lot

D. J. Haraway (2016, p. 55)

Software is now well represented in every nook and cranny of the world. Though this project directly engages software, software was also used at every stage of its production: conception, composition, distribution and (in the age of covid-19) discussion. In the same moment that our reliance on software reaches ever loftier heights, we are surrounded by stories of important software projects being biased against vulnerable groups. Standpoint theory suggests that such biases often come from the hidden and unacknowledged perspectives of the people making software (D. Haraway, 1988; Harding, 1992). In this sense, the problems caused by bias in software are not new problems. But, as Mackenzie (2006) and Kitchin and Dodge (2011) have noted, the management of a software project is filled with the work of managing relationships. Relationships between particular software artifacts (both the current version and older releases), the humans working on the project and expectations about the future. Software also exists within ecosystems: code written for a particular software project is surrounded and supported by code that many other software projects rely upon. How a particular software artifact impacts the world reflects the intent of both the organization that released this software artifact and the diffuse intent of the surrounding software/hardware ecosystem. Software interventions into existing projects can highlight the paths of agency within software artifacts. What can we learn about effecting change in software by studying how agency is mediated by the assembly process?

Software artifacts are the result of an assembly process that brings together the source code from many different software projects. Each software artifact is what Latour and Centre de Sociologie de L'Innovation Bruno Latour (1999) called a "blackbox" - a container that hides complexity. But in order to properly describe software, I needed to expand the idea of a blackbox. In Latour's theory, each blackbox contains other blackboxes with clean separations between those layers (Latour & Centre de Sociologie de L'Innovation Bruno Latour, 1999, p. 183). Software blackboxes, however, often interact across blackbox boundaries. A software artifact will use one library as a clean blackbox but interact directly with the inner-workings of another library. Exactly which pieces of code drive particular behaviors in a given software artifact is difficult to predict in advance and working backwards from artifact behavior to code identification is not straightforward. This be because a software artifact is a Composite - a heterogeneous mix of elements where the role of each element in any given software artifact is determined by the rules of the particular assembly process used to create it and whose component parts cannot be broken apart. This means that users and developers experience things in fundamentally different ways. Where users are always interacting with Composites, whose various component pieces cannot be differentiated within the Composite, developers are always interacting with pre-assembly components. This is because changing composites directly is inefficient, so modern software development process relies on making changes to components before one or more

assembly process(es). These components are where engineers leave imprints of their views. However, understanding how those views coe to the surface of Composites depends on examining the assembly process that creates each given Composite.The path of each materialized perspective through various assembly processes is unique.

Returning to the question of what one can do to address perspectives in software, this project takes the Free Open Source Software (FOSS) project Emergency Development ENvironment (EDEN) as its object of study and site of intervention. I chose EDEN because it is, as far as I can tell, a well-made piece of software that fulfills its role in the world. Others will engage socially problematic software and try to fix it, but the goal of my work here is to engage with a piece of software which is not obviously harmed by the standpoints of its developers. This does not aim to be a critical engagement, but an engagement whose successes and failures may be informative for future critical projects. The result of the modifications to EDEN can be seen at <url>. The changes to EDEN highlight where EDEN crosses blackbox boundaries, though my efficacy in making those crossings apparent is limited to particular *kinds* of boundaries. As we will discuss, software is deeply heterogeneous and that heterogeneity impacts the work that is needed to engage with it.

Software: a Thing and a Process

> *I have passed through a membrane where the real world and its uses no longer matter. I am a software engineer[.]"*
>
> *Ullman (2012, p. 3)*

It is difficult to talk with precision about what software 'is.' Individual programs that have been written in (or translated into) machine code are software (I will be calling these 'software artifacts'). Networks of software artifacts which only function when connected (such as the combination of one or more Facebook apps and the Facebook servers) also are clearly software. There are also software projects like the IETF Task Force that produce no software artifacts at all, but create and manage standards that are essential for the interoperability of software. The field is diverse and engaging with it demands some definitional boundaries.

Mackenzie (2006) focuses on understanding how 'software projects' function. The software artifacts produced by (or associated with) a given project, the people who work for the organization that owns the project, and imaginaries about the future or the purpose of the project are all part of Mackenzie's definition. A software project is made of individual artifacts (both software and otherz) and relationships. This means including both the particular qualities of each software artifact (how does the current version of a program work and how the next one *will* work) as well as the relationships between each software artifact (how the current version will give way [or not] to the next version) in software. It also means that software is the thinking, planning, and imagining about how the project will: work, relate to the world, and relate to other software artifacts and projects (Mackenzie, 2006). Software is both material and socially defined.

The qualities that exist within a particular software artifact, were first imagined

by engineers (much like Fleck (2012)'s thought collective) that work on that software artifact's software project or one of the supporting projects. The pipeline between how we imagine software will be in the future and the work of creating software that behaves like we imagine it should is important. It means that examining the details of how software works will tell us about how people imagined it would work.

The Material Character of Software

The material and social characteristics of software are co-constitutive its impact on the world. As Facebook attempted to respond the changing social situation around gender, they faced a back-and-forth between what was technically possible and generally allowed. They eventually settled on a solution that allowed lacuna to exist in the database in return for a more socially acceptable list of gender options (Bivens, 2017). Softwares flexibility means that nearly any situation can be represented and recorded, but each new piece of software is built on top of an ecosystem which makes certain things easy and other things hard. The database practices Facebook relied on in their early days represented gender as a binary value - a choice which later made it impossible to both offer non-binary genders and have a value in the gender field of the database.

Sociological work on new media supports the idea that there is something unique about how software impacts the world. Many of the ways we socialize through software have pre-digital antecedents, but the unique power of software to create new experiences is clearly visible. You do not have to be making software to feel the power *of* software to structure and channel your life. We are both using media in was that are only possible through digital systems and using digital media to account for ourselves in new ways and to new audiences (Humphreys, 2018; Jurgenson, 2019). Though we have always been different things to different people, it is only through software that we can be interacting with people who treat us as if we are famous and as if we are totally unknown simultaneously (Dean, 2010). The important thing to understand is that these new social realities reflect the new material force of software being brought to bear on society. These new bottles contain old wine, but it turns out that the bottle makes a huge difference to how we experience the wine. It isn't necessary for software to be attempting to transform the structure of society, its material force drives social transformation even when that transformation isn't imagined by its creators.

Software Cultures are Cultures of Materiality

Ethnographic work on communities that are deeply involved with software support the idea that the material qualities of software uniquely and meaningfully impact those who work on it. When Kelty (2008) argues that the open culture of digital spaces follows from the commitment to sharing, he is also suggesting a strong linkage between the material character of technology and the social goals of its builders. It would be possible to build software *wrong* and undermine the Free Open Source Software (FOSS) movements' goals (Kelty, 2008). His work underlines the importance of understanding the material qualities of software while it largely leaves them untouched.

The history of computers and computer work speaks to the material demands of software and the way that those material characteristics drive social factors. Computer workers gained their reputation for keeping odd hours and existing outside or alongside

the traditional power hierarchy because they needed to do work on mainframe computers when those computers weren't doing business calculations (Ensmenger, 2012). The particular materiality of computers at that time (that they were singular and could only run one task at a time) gave rise to social characteristics have have somewhat endured to today[1]. When Cox and McLean (2013) investigated the connections between speech and code, the material force of software was central to the connection. One reason code is special is that it is speech that can be mechanistically executed as well as being the content of a traditional speech-act.

There are even pre-computer arguments for seeing computer software as having unusual and deeply important material realities. Hacking is a cultural movement that began before software and which is focused on joyful play in eliding and subverting the intended purpose of objects (Drexler, 2019; Gabriella Coleman, 2012). That hacking has become so associated with digital technology testifies both to the power of the act and the complexity of the task. Hackers enjoy hacking on software because the medium is difficult and demanding. They can make a career out if it because of the enormous material impact of changing software from working *this way* to that way.

Materialized Perspectives

A model is worked, and it does work

D. J. Haraway (2016, p. 63)

My account of the pluripotency of software and its particular material impacts begs a question of the nature of software: how much of its impact is intended and how much of its impact is caused by bias within the software? Software is basically epistemic - it functions through selecting and acting on understandings of the world. Understanding bias and developing language around bias is a well-studied area in its own right. Bias can persist in knowledge and practice in a peripheral way (Fleck (2012) describes how syphilis retains a moral dimension even when it is understood to be a microbe) or in a way that determines the entire structure of thought on a subject (such as Said (1979)'s deconstruction of orientalism).

The theoretical framework I find most useful here is Harding (1992) and D. Haraway (1988)'s "standpoint theory." Specifically I'm interested in the connection (or lack of connection) between choosing one way of knowing or doing within software over another way of knowing or doing. When one does this without acknowledging the choice or connecting it to a wider world-view, it's what D. Haraway (1988) calls a god trick: the appearance of knowledge simply *being*, as if it came from nowhere and is not touched by the life background of the person who created it. God tricks are central to systems which embed unacknowledged standpoints within them. These systems can begin problematic, like how early biometric techniques tend to split populations into groups because they were built with a eugenic frame of reference Subramaniam (2014).

[1] I see a straight line between the historical social oddities of computer workers and the formation of the modern FOSS movement. Having a space where computer people already understood themselves as being "special" (even if it was special in an pejorative way) would have also helped them imagine that different property systems were possible. The early history of Linux was fueled by the unusual legal position of AT&T's Bell Labs (they were banned from making money off their work) which led to them taking few steps to protect their copyright and helping seed the open source movement (Kelty, 2008).

They can also accidentally become problematic, like how the fetishistic focus on the gene led us to over-invest in technologies that could not, on their own, provide us with new ways to affect the network of genetic and biological action that we call life Harraway (1997); Reardon (2017). Problematic characters of systems are allowed to stay hidden because the problematic qualities are not problematic for those who created the systems.

It is not necessary for a system to become problematic before it's useful to investigate standpoints encoded into it. Software, like any other tool, exists to help save human effort. In relieving humans from action, tools must trigger outcomes that might otherwise be proceeded by human choice. The number and nature of choices that can be elided within tools have only grown more diverse with time. The real world work of finding and addressing standpoints in problematic software can be assisted by finding and addressing standpoints that are not themselves problematic. When Traweek (2009) studied high energy physicists, her observation that the particle detectors built by different teams had characteristics that flowed from those teams' personalities and preferences did not need to be used to repair a flow to be useful. This project is interested in finding choices that software makes but does not acknowledge. These latter day god tricks escape from documentation or acknowledgment. They can be deeply aligned with their creators' world view and be difficult to describe to insiders. A great deal could be learned by engaging with the engineers who create these materialized standpoints, but we don't need to speak to them in order to point out what is there. ,

Software Without Ethnography

> *[T]he "writing of technology" is by by no means universal; the opaque and stubborn places do not lie simply beneath technology, but are wrapped around and in it*
>
> *Mackenzie (2006, p. 181)*

The bulk of sociological work around software has been work that with the material effect of software through its impact on social worlds. This project does not do that. It is conducted with the assumption that omitting social engagement will limit, rather than eliminating, the usefulness of the project. This belief is based on the studies of social outcomes that suggest that software possess a strong material character outside of social construction (Dean, 2010; Eubanks, 2018; Humphreys, 2018; Kelty, 2008). It is also based on work that show that once social information has passed into digital systems, that information will be manipulated and used in ways that can be driven as much by the quirks of digital systems as by social goals (Cheney-Lippold, 2018). This project, I believe, points towards a future project that engages more deeply in the social worlds of the makers and users of EDEN. Doing so would allow us to understand how EDEN constructs space for its makers, its users and the world in general (Kitchin & Dodge, 2011).

Better understanding the nature of software isolated from social situations will enable more complex work that studies both. It is those qualities of the digital world, what is found when you "pass through a membrane where the real world" no longer

matters, that this project is interested in finding (Ullman, 2012, p. 3).

Yes, but what IS Software?

> *[T]he "writing of technology" is by by no means universal; the opaque and stubborn places do not lie simply beneath technology, but are wrapped around and in it*
>
> *Mackenzie (2006, p. 181)*

Work on the powerful impact of software on our social world (and the power impact of our social world on software) somewhat elides the central question of my work: what *is* software? What are the qualities of this thing and this practice that has transformed our world, often on accident? Cramer (2008) focused on the Janus-like qualities of computer languages - forced to fulfill the needs of humans and machines. While Cramer (2008) notes that the only actual computer language are the machine languages spoken by computer chips, he also points out that the primary concern of general purpose 'computer'[2] languages (including python) is attending to different human priorities. This is especially true because all general purpose languages can be reduced to one another, so any difference in functionality come only from the difference in how well the language suits the preferences of the human writing it. Yuill (2008) talks about the character of the ecosystem of programs and the centrality of "interrupts" to modern system design. Most software written today is written with the expectation that it may be interrupted. This is because modern computers remain responsive to people (and other software artifacts running on the computer) by being ready to interrupt any process at any moment. This is important because it prepares software to try to deal with the unexpected.

I find Latour and Centre de Sociologie de L'Innovation Bruno Latour (1999)'s blackboxes a perfect fit[3] for talking about software and how it functions in the world. Blackboxes are enclosed units that accomplish a task without exposing outsiders to the interior details of how it works. Like software, blackboxes can contain almost anything with any level of complexity, but to those outside the box it appears singular and cohesive. It's only in failure that blackboxes are differentiated from other objects. Latour was interested in how, once a blackbox failed, the fact that it has components becomes socially available and socially unavoidable. Often, blackboxes contain other black boxes, each one waiting for failure to make the fact that it has internal components. Software too, has important internal components that are invisible. However, because the tools and skills needed to disassemble software are so much rarer

[2] 'General purpose' programming languages are all turing complete, which means they can fully describe the behavior of a 'turing machine' - a theoretical automaton described by Alan Turing in the early 20th century. This means that all general purpose programming languages can be reduced to one another (a series of instructions generated by one language can be generated by another). However, there are also 'domain specific languages' that are designed to solve more specific problems that cannot be reduced to one another (and cannot fully describe the behavior of a turing machine). Ironically, this document was typeset in LaTeX, which is turing complete.

[3] Computer science draws on the general blackbox concept that Latour drew inspiration from to talk about acting on formal function definitions without knowing about how the function is implemented (Abelson & Sussman, 1996, p. 33).

than the skills and tools to disassemble, in Latour's example, a projector there are important differences in how software is perceived socially.

Taking Action

> *We live in capitalism, its power seems inescapable – but then, so did the divine right of kings. Any human power can be resisted and changed by human beings.*
>
> *Le Guin (2014)*

So in one hand I hold Latour and Centre de Sociologie de L'Innovation Bruno Latour (1999)'s blackboxes and in the other Harding (1992) and D. Haraway (1988)'s situated knowledges. As Mackenzie (2006) recounts, the process that produces software is complex and heterogeneous. It defies simple analysis and it, itself, a deeply social thing that will also produce software artifacts that can be mechanistically executed (Cox & McLean, 2013). Yet we are beset on all sides with software whose bias is plain for all the world to see (Bivens, 2017; Dean, 2010; Elish, 2019; Eubanks, 2018; Schüll, 2012). How can our academic theories be used to address these practical concerns? How can we find what Sismondo (2008) called an "engaged program," which combines theory and contact with the real world?

The first step to to try doing real work on real software. Zuiderent-Jerak (2015)'s *Situated Intervention* (which also draws from situated knowledges) argues forcefully that social scientists can and must intervene to learn. He describes how medical professionals use interventions (and their outcomes) as tools to generate knowledge and test assumptions. My own pre-academic experience as a software engineer supports the knowledge generating power of careful and measured intervention. This project hopes that it can find materialized perspectives within a wholly unobjectionable software project (and its associated software artifacts) and, though engaging with the source code of that software project, learn something about how we might address materialized bias in more convoluted situations.

The object of study

EDEN

> *That virtual worlds are places means they can be fieldsites;*
>
> *Boellstorff (2015, p. 107)*

The particular piece of software this project studies is Emergency Development ENvironment. EDEN was originally created by a coalition of Sri Lankan Information and Communications Technology companies in the wake of the 2004 Indian Ocean Earthquake and Tsunami. The project is now managed by the nonprofit Sahana Foundation. It has been used in numerous disaster responses since 2004 and is also used by a number of organizations for managing resources outside of any specific disaster (Sahana Foundation, n.d.). EDEN's functionality can fairly be described as tracking and managing resources. Its power comes from the enormous range and detail of the information it knows how to track. EDEN has indexes for organizations, people,

projects, events, facilities, supplies, documents, possible scenatios, and events. It also provides on-platform messaging and special tools for the management of emergency shelters (Sahana Foundation, 2011). Each resource is meant to be connected to other resources: people work for organizations, projects are run by organizations and are associated with people, facilities are linked with projects and organizations and other resources like supplies and verticals. The flexibility of the software means that it can be used to manage a single organization, multiple organization, or used as a hub for many different groups coordinating around shared goals but without central authority. The system aims to be as comprehensive as possible and to use the same entity for resource and system management. The entries for people can act as a simple rolodex and will also double, if desired, as that person's account within EDEN. The interface for managing goods is the same interface for managing access to the management system. To the degree possible, EDEN is structured to allow the same people who do the day-to-day work of the organization to administer the system that manages the organizations' resources. It also aims to be an effective tool for people at all levels: administrators can track where people and supplies are allocated and volunteers can access their assignments, documents and information about how to use resources.

Why EDEN

I selected EDEN for this project for two primary reasons: it is a Free Open Source Software software project in a language I am familiar with and it is not obviously problematic. I will detail its open source pedigree shortly, but first I want to explain my second selection criteria. "Improving" software (whatever improving means) can usefully be separated into at least two questions: what quality should change and how do you bring that change about? This project attempts only to speak about the second (likely easier) question. Partially this is because the project was designed to fit within the confines of a final project for a masters degree. This project represents, roughly, three graduate level classes worth of work. I didn't feel like I had time to find *and* fix flaws. So this project should be understood as neutral on the question of, "is EDEN good?" Nothing about EDEN seems bad to me. They have received numerous awards, glowing testimonials, and are used by many large organizations that could use other products if EDEN were lacking.

I lack the experience and expertise to say that I think EDEN is *good*. Assessing if software is "good" or "bad" is not straightforward. Simply examining software artifacts in isolation tell us very little. It's only through engaging with one or more software artifact(s) as they exist in the lived world and contextualizing that with detailed ethnographic work that it's possible to start making value judgments (Eubanks, 2018; Schüll, 2012). Those judgments wouldn't be universal, of course, but would be about a particular population. So this paper is done from the perspective that EDEN seems fine to me. The project is interested in the technical details of how the EDEN software artifact emerges out of its Software-Object and how agency is modified by that process.

Finally, and briefly, this project does not take the position that the technical qualities it investigates are free from the impact of social structures. Gabriella Coleman (2012) and Kelty (2008) have both compellingly shown that the social and technical co-produce each other. However, we can still usefully speak about qualities that technical systems have and how those qualities impact our lived experience. That this

project has obvious extensions in the social and ethnographic realm is a strength and declining to investigate them should be understood as a concession to time.

Out of Many, One

Wherever possible, EDEN uses FOSS technologies. The language it is written in, the libraries it relies on to provide functionality, the tools it uses to support its functionality, and its main operating system are all both open source and available at no cost. EDEN will generally operate on top of non-FOSS systems like Microsoft Windows, but the team doesn't prioritize systems outside of the FOSS systems they develop and test on (Sahana Foundation, 2015). Kelty (2008) talks about how FOSS is both a philosophy and a system of development that has practical impacts. One of the side effects of EDEN committing to use the FOSS ecosystem is it makes my form of engagement possible. Though it is possible to examine compiled machine code and draw some conclusions about the intent and process that assembled it, such a project would be far outside my capabilities. Instead, the source code of EDEN, web2py (the web framework EDEN uses), Python (the language EDEN and web2py are written in) and all of the libraries used by the project are open source. Their preferred databases (MySql or PostgreSQL) are also fully open source projects. Open source projects often don't just publish the current source code, but offer full histories of what change, when it changed, who changed it and how it was changed. These changes often include notes about why *that particular* change was made over any other possible change (Chacon & Straub, 2014, p. 13-16). Open source projects also commonly have systems for tracking lists of unfixed flaws as well as planned future improvements (both types of Mackenzie (2006) relationships), but those systems operate above the layer of source code so this project does not engage with them.

Investigating EDEN

My work began, as all software work does, with work of assembling the components that transform EDEN from a collection of python source files into a software artifact. This requires connecting a number of components together: libraries and tools in the glspython language and a wider set of heterogeneous tools (databases, non-python libraries and tools). Some of these components remain as source code to be interpreted by python, some of them are downloaded as source code and go through their own assembly process to generate their own software artifacts. The assembly process that draws the components of EDEN together does not happen in a single step, but is a number of loosely ordered heterogeneous processes. Some must be completed before others and there is an officially recommended order in which to prepare the various components, but often these orders are more matters of preference than necessity. What is important is that all of the required components are collected in their expected places by the time EDEN itself is run.

The Sahana Foundation recommends that developers use a Virtual Machine that has EDEN and its supporting libraries installed through a script. However, at the time that I engaged with the project, the instructions for how to install EDEN focused on using operating systems and python versions that have since stopped being updated[4].

[4] As Mackenzie (2006) points out, software is a process not a thing. This is especially true for software like EDEN, which is made available to users over network connections. Pieces of software that stop

Though more modern scripts can be found, the standard install scripts for the EDEN Virtual Machine and the script to install the software on a physical machine both use operating systems released in the mid-2010s that have not been maintained for at least 5 years (Canonical, 2020). However, EDEN is aware that this is problematic and makes an effort to support newer versions of python and newer operating systems, but official instructions to use the latest software are sparse.

Approaches to Tooling

Choosing to offer a fixed and unusually old set of tools is the first materialized I found encoded into EDEN[5]. The state of the scripts reflect a culturally important choice. The team could note and situate their approach in their documentation, but they have not. However, project-wise choices to use older or newer versions of tools are common. There are software projects (sometimes entire programming languages) that greatly value using the most up to date version of a tool or library (Fernandes da Costa, 2017). These communities have legitimate fear that older versions of tools are examined less often and will likely be fixed less frequently. They feel that using the latest (within reason) version is a goodness in of itself[6]. If the EDEN team followed this philosophy, they would not default to using older operating systems and versions of python. They would need to watch their ecosystem of tools and libraries more closely, but the versions they were watching would also be more in the collective consciousness of the FOSS world.

All this isn't to say that the EDEN team choice isn't a sensible one. The latest versions of tools often lose track of the world outside the top of the capitalist pyramid.

---

receiving updates can quickly become vulnerable to new interactions with an ever-changing ecosystem. These vulnerabilities can result in programs becoming unstable, new security vulnerabilities, or simply an inability to keep up with the changes to other components. Software that is no longer being "maintained" (the phrase used to describe engineering work to patch security holes, update interfaces to modern standards, and make those updates generally available) can suddenly stop working because one or elements of its ecosystem cuts off old software. It is true that isolated software can run for years or decades without changes and that, for some systems, users never experience adverse consequences for leaving them frozen in time. However, whether or not a program can "safely" be left alone is highly dependent on external factors and software that seems "safe" from needing changes can suddenly and tragically need emergency updates at high cost (Lee, 2020).

[5] It's a fair question to ask where EDEN begins and ends. Certainly its source code is part of the program, but I would also include the collected documentation on how that source code should be deployed and supported as part of the "program," even thought it's not actively involved in the functioning of an assembled software artifact. This is best understood with Star and Griesemer (1989)'s idea of boundary objects - coordinating objects that are able to play multiple roles for different actors. The software artifact of EDEN cannot exist for anyone outside of its immediate development team without distributing instructions on how to assemble the software artifact, so I include that material in the EDEN boundary object. It's worth noting that many of the requests for help with EDEN on the Sahana Foundation mailing list are questions not about EDEN per-se, but about the tools and libraries that support a fully functioning EDEN. In a very real sense, the EDEN that exists without instructions on how to assemble it isn't available to me: I wouldn't know how to access it.

[6] A language has evolved to speak about the relative stability of a particular software artifact (or, more precisely, the boundary object that includes a software artifact and its assembly instructions). Signaling about how often you will be changing your software allow users to choose the version of you work that will fit their own rhythm (Canonical, 2017). 'Unstable' releases are often more likely to have bugs or be insecure, but also have new features that cutting-edge projects may need. 'Stable' releases have few new features but are well understood and unlikely to be the source of problems.

The same networks of power that structure the spread of physical technology also shape the attention of active software engineers. EDEN has chosen to use established and well-supported versions of their tools[7]. Their older set of tools is more likely to have undiscovered (or unpatched) security vulnerabilities, but they are virtually certain to function properly and immediately allow the successful assembly of an EDEN software artifact. That set of probabilities seems very well suited to the expected audience for individuals first setting up their own copies of EDEN: individual developers setting the system up to better understand it. When another team decides to deploy their own copy of EDEN in a way that's available to the public, they can assemble that software artifact using a balance of newness and stability that feels comfortable to them[8].

Investigating Assembly

It's one thing to identify where the EDEN install scripts sit in a cultural gradient. It's another to engage the particular choices that the EDEN project has made. I chose to install a far newer set of tools, based on the projects newest recommendations for new installs (König, 2019a). I was guided by the python 2.7 installation script provided on the Sahana EDEN wiki[9]. However, I chose to use a version of python originally released in 2016 (six years after the default version used by EDEN). Using a newer version of python seemed like a sensible software engineering choice and also a way to check if EDEN was, as they claim, able to operate equally well on the version I selected (3.6.9) and the 2.7 version that EDEN uses by default[10].

Selecting a more modern python turned out to have knock-on effects. Some of the libraries that EDEN requires are written only for python 2.7 and replacements were needed to function with 3.6. One of the major changes was that the version of Web2Py the EDEN scripts installs does not support python 3.6 and so I needed to chose an updated version. I selected 2.18.5, but this new combination of the EDEN source code and the Web2Py source code revealed that the two projects had drifted apart in small but essential ways. Specifically, in a section of the Web2Py and EDEN code that handled data validation.

Data validation is, in general, the work of verifying that information conforms to a set of standards and informing the user how it has failed to conform when a problem is

---

[7] As of the writing of this paper, the versions of Ubuntu that EDEN installs into by default have left long term support. Though evaluating if that's good or bad is outside the scope of this paper (and my expertise) it seems important to mention it.

[8] Of course, this update process also has risks. A well known mix of library versions will work together, but as I'll note later, mixing different versions of libraries can cause unexpected dysfunction. Suffice to say that the old versions of the tool and library ecosystem will generate the fewest unexpected tooling problems and maximize for developing in an environment familiar to the EDEN team.

[9] The Script I used can be found `http://eden.sahanafoundation.org/wiki/InstallationGuidelines/Linux/Server/CherokeePostgreSQL`, though as I describe, I often used different components. Unedited notes on the things I ran into can be found at `https://github.com/aeturnum/masters_project/blob/master/Notes/Web2Py_Notes`.

[10] I selected a release from the 3.6.X series because I'm fond of f-strings (a features added in 3.6 that allows compact formatting of printed information). However, 3.6.9 is now actually out of date, as 3.6.10 was released just a few days before the end of 2019 (Python Software Foundation, 2019). In a perfect world I would have used 3.6.10, but as this project includes no private information I stuck to 3.6.9.

found. This should always be performed any time data is stored in a system. Digital storage systems have expectations that, when violated, can cause immediate systems failures. Web frameworks commonly provide facilities for verifying data and Web2Py is no exception. Web2Py has components that will check that that user input can be converted to numbers or dates or email addresses, among others. Web2Py is open source and so all of that functionality can be expanded by programs that make use of Web2Py and EDEN does so in a number of places.

Shifting Relationships

In between the version of Web2Py that EDEN uses by default and the version of Web2Py that I selected, the control flow of data validation has changed slightly. This change meant that custom data validation classes that were written using the older Web2Py system no longer fit into the expected control flow of data validation. The EDEN data validation code, based on the older standard, is not called properly when using the newer version of Web2Py. This doesn't stop the successful assembly of an EDEN software artifact, but once that software artifact is assembled most of its functionality in inaccessible because the EDEN-specific data validation code won't run and so the system won't accept that data.

This problem is interesting because it turns out it captures a moment in time between the Web2Py team and the teams that use Web2Py (i.e. EDEN). If I had selected a slightly earlier version of Web2Py, then it would have used the old system and the EDEN code would remain functional. Web2Py immediately realized the problem and integrated a fix[11] for the issue. This shows the management of relationships that Mackenzie (2006) describes. There is the work of making software and then the subsequent and parallel work of adjusting that software to stay within relationships boundaries. The Web2Py successfully implemented a new way of handling data validation control flow, but they neglected that they were doing it in such a way as to break an existing relationship they did not mean to break. Because I happened to select a version of Web2Py that was temporarily out of relation with EDEN, I failed to assemble a fully functional software artifact. This was what got me interested in the assembly process itself.

Investigating Assembly

Assembly is a label I'm using to cover a variety of software processes that source code (and many other things) and produce software artifacts. Almost all software is written in programming languages that have the dual goal of being possible for humans to understand and being suitable for mechanical translation into machine code. Machine code is, after all, the only language a computer can really understand (Cramer, 2008).

Methods of assembly vary widely, but they can roughly said to lie between compilation and interpretation. The purest compiled language will generate a software artifact that contains all the machine code that will be used by the software artifact into a single contiguous piece of machine code. That machine code will generally only

[11] Interestingly enough, the problem was fixed by a piece of code written by a member of the EDEN team (König, 2019b). The Web2Py team had already noticed the problem and thought they had fixed it, but their fix did not work for for all circumstances and required more work.

work on a single hardware architecture, but the same source code can be assembled[12] into machine code on other architectures by different assembly toolchains. On the other side of this spectrum are interpreted languages. A purely interpreted language assembly process does not involve the generation of any new machine code. Instead, it uses a standard software artifact (called an "interpreter") that reads the source code of the interpreted program in its human readable form like a recipe: reading the file(s) as the interpreted program executes and using machine code that exists within the artifact of the interpreter to tell the hardware what to do.

There are also a variety of methods for software artifacts to share access to centers of functionality. The most common is sending calls "out" to libraries. The pure compilation situation described above really only exists on hardware without a multitasking operating system. For every software artifact on your phone or computer, the software artifact will frequently make a call to a library (which itself sits somewhere between compiled and interpreted) where it specifies which library, which section of the library, and the information it would like to pass off. Then the operating system pauses the execution of the machine code in the software artifact and loads the library (or the library's interpreter) to perform the requested task. When the task is done, the first software artifact is loaded again and told the result.

Most languages and software artifacts sit somewhere between being fully compiled and fully interpreted. Python is largely an interpreted language, but it also creates non-human readable bytecode versions of each python source file that allow the interpreter to execute more quickly (The Python Software Foundation, 2020). Python also allows access to libraries written in machine code and EDEN invokes machine code libraries through python interfaces.

Digital Relationships

These libraries exist on the border of all of the software artifacts that use them[13]. They contribute to how the software artifacts that use them behave. Flaws in libraries can allow the programs that use them to be taken over. Sometimes these flaws only require attention to discover, but it's often the case that other elements of the hardware / software ecosystem must change before they can be found (Huang, 2003). All this is to say that the kind of assembly problem I encountered in assembling EDEN is not particular to EDEN. It's a universal concern when assembling software artifacts that use any functionality that resides outside their own source code (which nearly all do).

This system of linking together different code written by different people is an example of the relationships that Mackenzie (2006) found at the center of the software projects he studied. At the project level this means that a the people working on one

---

[12] The assembly process for a compiled language (and in fact for many languages that sit somewhere in the middle of the compiled-interpreted range) is generally executed by a software artifact called a "compiler." I'm avoiding using the common word to avoid confusion in with the range of approaches in assembly.

[13] Libraries do not have to exist outside of machine code. When a library is 'statically-linked' to a software artifact, the section of the library's machine code that is used by the software artifact is copied into the software artifact's machine code. So when the statically-linked assembled software artifact calls the statically-linked library, the computer simply loads the section of the library that was copied into the software artifact code.

project must pay attention to the development of many other projects. Are new versions of the libraries that you use being released? Will the old version be updated if security flaws are found? Does the new version have features or changes that would be useful for our project?

What is striking about the problems encountered while trying to assemble my own artifact of EDEN is that, while every individual who assembles an EDEN software artifact will need to go through their own process, the problems I encountered are particular to using an un-patched version of Web2Py 2.18.5. The install script I used contains other fixes for other aspects of that particular pairing of components, as well as different fixes for different combinations of components (Sahana Foundation, 2018). EDEN can be assembled using an extremely wide set of components. It can run on that haven't been considered up to date for over ten years ago and components that are still considered cutting edge now (as well as many mixes between these two extremes). The EDEN software artifact is designed with this capability in mind. It is a strength of the FOSS ecosystem that such a wide range of compositions is possible, but it also means that EDEN can be a nearly limitless number of different software artifacts whose qualities emerge both from EDEN's source code and the interactions between the particular versions of the components.

The Social and Material Components of Software

Keeping in mind the range of possible component sets for EDEN, think back to the particular set used in the install script. I speculated the set in that script were selected because they're well known familiar to the EDEN development team. They minimize the chance that people will mistake a problem caused by assembly for a problem that exists within the EDEN source code.

When a particular software artifact fails to function as expected, there are many possibilities as to why. It might be mis-configured, the resources it needs to function properly might not be available, there may be an internal incompatibility between components, or there may be an error in the EDEN source code. There are many functional EDEN software artifacts that contain component combinations that the developers have never tried and have no experience with. When what appears to be an error is reported to the EDEN team, often the first questions they ask are about the versions of the libraries that were involved. If the versions of the libraries being used aren't considered to be the correct ones, the bug can be closed as wholly or partially invalid (trendspotter, König, & Boon, 2020).

This method of engagement around questions of proper assembly and proper component selection are, above all, practical. Over time the number of possible combinations of all the versions of all the libraries will inevitably expand beyond the abilities of any team. To control this somewhat, teams will have defined periods of support and lists of appropriate versions of components and only fix problems that found within that somewhat limited domain (Canonical, 2020; König, 2019a). But software artifacts that are assembled using components drawn from outside the official set of components may still function. Being 'out of support' does not mean that things will not work. It means that the team managing the software project has no opinion about the relationships between their code and the components you're using. They will support the official supported combinations and leave you to do your best with the

unofficial ones.

This suggests a limit on the boundaries of the EDEN software project. Maybe the unofficial software artifacts that the EDEN team won't support now aren't really EDEN software artifacts. However, I don't think it's that simple. What components (and versions of the EDEN source code itself) are inside the 'official' EDEN change over time. What is an unofficial EDEN software artifact might one day become an official software version as the EDEN software project changes its thinking about relationships. By the same stroke, official versions will often become unofficial. The team recently announced that, with the end of official support for python 2, they will be moving to using python 3 for all of their builds and updating their install scripts to use python 3. Suddenly, my EDEN software artifact, which already uses python 3 is within the official definition and the EDEN software artifact assembled by the script on the Sahana website is unofficial (König, 2019a; Sahana Foundation, 2015).

Software Exists Outside of the Social

Instead, I think what this shows is a material reality interacting with a social relationship. Each given version of EDEN's source code has a particular material relationship with each version of each component it could use. These relationships have an existence outside of any social system because computer code can be executed mechanistically (Cox & McLean, 2013). The official / unofficial layer, where the team that writes the source code for EDEN and sets forth the list of officially supported component relationships, exists on top of this material reality. But this social layer does not change what software *is*.

Understandings of software where only the official versions of EDEN are "the software called EDEN" lead to an understanding of software that defies common experience. It would mean that, once the EDEN team announced that they were migrating away from python 2, all of the python 2 EDEN software artifacts that exist and function now suddenly stopped being software. It would also mean that, because I have changed some elements within the EDEN source code[14], my software artifact can no longer be EDEN - even if I used all the official supporting components. It would require an impossible level of information about every component and every element of every software artifact that appears to be EDEN.

Once we accept both official and unofficial software artifacts are that software, it becomes clear that another social understanding exists along side just the official and unofficial version of software. Most people who encounter a particular software project have no idea if they are interacting with an official or unofficial version[15] (or that such a distinction even exists). So there exists this general social understanding about each

---

[14] Modifications to the EDEN source code are also common. I've modified the eden source code somewhat and it is, I would argue, still EDEN. Where one draws the line is questionable and, I think, worth studying in its own right.

[15] It is easy to example how someone might assemble an unofficial version of a piece of open source software like EDEN. It may seem harder to imagine how the public gets access to an unofficial closed source piece of software that's distributed as machine code (like, say, the Facebook app). However, this situations are not as rare as you might expect. If you've ever owned a phone that is no longer receiving updates, or used a version of Windows that Microsoft no longer supports, every program in that environment is likely unofficial - because the companies no longer support that environment.

piece of software (including EDEN) that individual draw from their own experiences. These experiences might be with official software, unofficial software, or even be based on a mis-identification of the software (such as receiving a counterfeit product and leaving a bad review for the original) (Suthivarakom, 2020).

It's useful to draw out these different social experiences of software because it allows us to think in a more informed way about the impact of materialized perspectives. It is not just the source code of a particular software project that adds to its behavior, it is also the various components that source code is combined with. For those who only interact with post-assembly software artifacts, the relationship of the software they are using to the official software is unknowable. Software that has the reputation for being unstable and unreliable can get that reputation because people are, knowingly or unknowingly, using pirated copies that crash because of anti-piracy code (Fitch, 2008). The team that makes that piece of software may never know why people think its unstable, because they would need to understand the assembly processes that particular software artifacts went through. The users may want the software to be different, but though the software team is the single entity most responsible for how a particular software artifact functions, they may simultaneously be unable to understand or repair the problem.

Glossary

Composite A Latour and Centre de Sociologie de L'Innovation Bruno Latour (1999)ian black box whose functionality cannot be reduced to a blackbox containing other blackboxes. Instead, a composite is a network of blackboxes are heterogeneously linked. In addition, where the contents of a blackbox become socially visible when the blackbox stops working, composites can totally fail and remain opaque.. 3, 4

python Python is an open source, interpreted, high-level, general-purpose programming language. It was initially released in the early 1990s, but has been continuously updated since then. Python 2 was the standard from 2000 to 2008, but in 2008 Python 3 was released with backwards-incompatible changes. This split the Python community, leading to Python 2 and Python 3 having slightly different versions of tools available. Python 2 continued to receive support and updates until 2020, when official support for Python ended (Wikipedia contributors, 2020). The open source nature of Python means that it is likely that Python 2 will continue to be updated by users after official support ends. . 8, 11, 12, 13, 15, 17

Sahana Foundation The nonprofit that manages the EDEN project. Based in Los Angeles with an international scope and focuses on supporting communities in disaster preparedness and response through information technology.. 9, 11, 12

Software-Object A generic term that describes the entire process that leads to the production of a series (or a single) software artifact. This includes the economic and cultural position of the organizing project, the social organization of those who write the code for the piece of software, the social positions of the software artifacts' users, and the software artifacts themselves. This term is used in

opposition to the common "software" because of the potential confusion between a "software project" (the organizing structure of a Software-Object), a "program" (a Software-Artifact), and other linguistically confusing ways of speaking about the multi-layered process that produces "software.". 10

software artifact An entity of uncertain composition that behaves, for common users (as opposed to technical staff) as a single entity. A software-artifact could be a single binary with all its functionality provided by machine code included in that single location in memory. It could also be a binary that makes calls to various libraries. In the case of EDEN, it's an expensive set of Python files - some of which are human readable and some of which have been modified to make them faster to read. In this case a separate program, the Python binary itself, reads and executes the EDEN program. Even though EDEN is many separate files and contains no machine code, it is properly considered a software-artifact because users interact with it as a single entity. Software-artifacts are produced periodically by software projects, but are not themselves entirely "software." The idea of software must be large enough to allow multiple software-artifacts to exist within the same project: the current version of your phone OS and the next version, the Facebook app on your phone and the web server that tells it what has happened. Nevertheless, when we are using "software" we are always directly interacting with one or more software-artifact(s). They are the material reality of software projects.. 3, 4, 5, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18

Virtual Machine A family of packaging techniques that allow packaging and distribution of an entire computing environment (operating system, libraries, programs) in a single digital object. This allows one physical computer to run several independent "machines" simultaneously. When a virtual machine is started, it the hardware it sees is virtual hardware simulated by the program "hosting" the virtual machine. The hosting program then translates requests to that virtual hardware into requests to the real hardware and relays the responses. Running a program in a virtual machine is slightly less efficient than running it on real hardware, but this cost is often less than the cost of manually arranging non-virtual installs on real machines.. 11, 12

Web2Py An open source web framework for Python 2.7 and 3+.. 13, 14, 16

web framework A heterogeneous set of software that allow a programmer to efficiently write and manage providing a service over the Internet. This could be a website or a mobile app (often it is both) or a go-between for other services. Examples include web2py and Django in Python or Phoenix in Elixir.. 11

Acronyms

EDEN Emergency Development ENvironment. 4, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18

FOSS Free Open Source Software. 4, 5, 6, 10, 11, 12, 16

ICT Information and Communications Technology. 9

References

Abelson, H., & Sussman, G. J. (1996). *Structure and interpretation of computer programs* (2nd ed.). Cambridge, MA, USA: MIT Press.

Bivens, R. (2017, June). The gender binary will not be deprogrammed: Ten years of coding gender on facebook. *New Media & Society*, *19*(6), 880–898.

Boellstorff, T. (2015). *Coming of age in second life: An anthropologist explores the virtually human*. Princeton University Press.

Canonical. (2017, March). *LTS.* `https://wiki.kubuntu.org/LTS`. (Accessed: 2020-5-7)

Canonical. (2020, April). *Releases.* `https://wiki.ubuntu.com/Releases`. (Accessed: 2020-5-7)

Chacon, S., & Straub, B. (2014). *Pro git*. Apress.

Cheney-Lippold, J. (2018). *We are data: Algorithms and the making of our digital selves*. NYU Press.

Cox, G., & McLean, C. A. (2013). *Speaking code: Coding as aesthetic and political expression*. MIT Press.

Cramer, F. (2008). LANGUAGE. In M. Fuller (Ed.), *Software studies: A lexicon* (pp. 168–174). MIT Press.

Dean, J. (2010). *Blog theory: Feedback and capture in the circuits of drive*. Polity.

Drexler, D. (2019, March). *Hack the planet.* `https://medium.com/@aeturnum/hack-the-planet-aaa4abc23bc8`. Medium. (Accessed: 2020-4-18)

Elish, M. C. (2019, March). Moral crumple zones: Cautionary tales in Human-Robot interaction. *Engaging Science, Technology, and Society*, *5*(0), 40–60.

Ensmenger, N. L. (2012). *The computer boys take over: Computers, programmers, and the politics of technical expertise*. MIT Press.

Eubanks, V. (2018). *Automating inequality: How High-Tech tools profile, police, and punish the poor*. St. Martin's Press.

Fernandes da Costa, L. (2017, February). *Understanding dependency management in go.* `https://lucasfcosta.com/2017/02/07/Understanding-Go-Dependency-Management.html`. (Accessed: 2020-5-8)

Fitch, M. (2008, February). *Venting my frustrations with PC game-dev.* `https://forum.quartertothree.com/t/venting-my-frustrations-with-pc-game-dev/42627`. (Accessed: 2020-5-24)

Fleck, L. (2012). *Genesis and development of a scientific fact*. University of Chicago Press.

Gabriella Coleman, E. (2012). *Coding freedom: The ethics and aesthetics of hacking*. Princeton University Press.

Haraway, D. (1988). Situated knowledges: The science question in feminism and the privilege of partial perspective. *Fem. Stud.*, *14*(3), 575–599.

Haraway, D. J. (2016). *Staying with the trouble: Making kin in the chthulucene*. Duke University Press.

Harding, S. (1992). Rethinking standpoint epistemology: What is" strong objectivity?". *Centen. Rev.*, *36*(3), 437–470.

Harraway, D. (1997). *Modest_Witness@Second_Millennium.FemaleMan©_Meets_OncoMouse*^{TM}. Routledge New York.

Huang, A. (2003). Keeping secrets in hardware: The microsoft XboxTM case study. In *Cryptographic hardware and embedded systems - CHES 2002* (pp. 213–227). Springer Berlin Heidelberg.

Humphreys, L. (2018). *The qualified self: Social media and the accounting of everyday life*. MIT Press.

Jurgenson, N. (2019). *The social photo: On photography and social media*. Verso Books.

Kelty, C. M. (2008). *Two bits: The cultural significance of free software*. Duke University Press.

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

König, D. (2019a, December). *End of python-2.7 support in sahana eden.*

König, D. (2019b, April). *pydal@cecd771.*

Latour, B., & Centre de Sociologie de L'Innovation Bruno Latour. (1999). *Pandora's hope: Essays on the reality of science studies*. Harvard University Press.

Lee, A. (2020, April). Wanted urgently: People who know a half century-old computer language so states can process unemployment claims. *CNN*.

Le Guin, U. (2014, November). *Medal for distinguished contribution to american letters acceptance speech.* National Book Awards.

Mackenzie, A. (2006). *Cutting code: Software and sociality* (S. Jones, Ed.). Peter Lang Publishing.

Python Software Foundation. (2019, December). *Python release python 3.6.10.* `https://www.python.org/downloads/release/python-3610/`. (Accessed: 2020-5-10)

Reardon, J. (2017). *The postgenomic condition: Ethics, justice, and knowledge after the genome*. University of Chicago Press.

Sahana Foundation. (n.d.). *Making chaos managable.*

Sahana Foundation. (2011, December). *Sahana eden brochure.* Online.

Sahana Foundation. (2015). *InstallationGuidelines/Windows.* `http://eden.sahanafoundation.org/wiki/InstallationGuidelines/Windows`. (Accessed: 2020-5-6)

Sahana Foundation. (2018, February). *InstallationGuidelines/Linux/Server/CherokeePostgreSQL.* `http://eden.sahanafoundation.org/wiki/InstallationGuidelines/Linux/Server/CherokeePostgreSQL`. (Accessed: 2020-5-23)

Said, E. W. (1979). *Orientalism*. Vintage Books.

Schüll, N. D. (2012). *Addiction by design: Machine gambling in las vegas*. Princeton University Press.

Sismondo, S. (2008). Science and technology studies and an engaged program. *The handbook of science and technology studies*, *3*, 13–32.

Star, S. L., & Griesemer, J. R. (1989). Institutional ecology, translations' and boundary objects: Amateurs and professionals in berkeley's museum of vertebrate zoology, 1907-39. *Soc. Stud. Sci.*, *19*(3), 387–420.

Subramaniam, B. (2014). *Ghost stories for darwin: The science of variation and the politics of diversity*. University of Illinois Press.

Suthivarakom, G. (2020, February). *Welcome to the era of fake products.* `https://thewirecutter.com/blog/amazon-counterfeit-fake-products/`. Wirecutter. (Accessed: 2020-5-24)

The Python Software Foundation. (2020, May). *Glossary.*

`https://docs.python.org/3.6/glossary.html`. (Accessed: 2020-5-22)

Traweek, S. (2009). *Beamtimes and lifetimes*. Harvard University Press.

trendspotter, König, D., & Boon, F. (2020, April). *Several various NoneType errors (setup, vol, project, inv, dc, ...)*. `https://github.com/sahana/eden/issues/1543`. (Accessed: 2020-5-24)

Ullman, E. (2012). *Close to the machine: Technophilia and its discontents*. Picador.

Yuill, S. (2008). INTERRUPT. In M. Fuller (Ed.), *Software studies: A lexicon* (pp. 161–168). MIT Press.

Zuiderent-Jerak, T. (2015). *Situated intervention: Sociological experiments in health care*. MIT Press.