The Poverty of Post-Open Source

Gnuxie Lulamoon and Hayley Patton

September 4, 2021

Contents

One thing which has bothered us is that there seem to be similar themes in prior writeups – in a way, it appears that writings like Terminal boredom, Ethical software is a sad joke, Careful with that axe, Eugen and A Parastatal Problem approach symptoms of the same cause. So it is natural, in a way, that we want to tie the loose ends between them.

Much like the phantasm of The Far Left which scares the life out of the average conservative, the idea of a unified "ethical software" or "cooperative software" movement is completely imaginary. In a prior article, we contrasted our own ethics to "vulgar ethical software". Noting that proponents of this vulgarity often claim inspiration from an article entitled Post-Open Source, we now consider vulgar ethical software to more accurately be the lineage of this article; as they don't even call their ideas "ethical" to start with.

The only actual similarity between our theory, which we half-jokingly call *egoist telekommunism*, and the lineage of this *post-open source* is that the authors of the original texts of telekommunism and post-open source both like Karl Marx. So, the purpose of this article is to discuss what isn't similar, and why we think the telekommunist approach is much better.

The user is always an idiot

A general theme in post-open source is it assumes the same dichotomies and distinctions between producer and consumer, or between programmer and user, from a capitalist frame of view. From the start of Post-Open Source:

> but those freedoms don't actually mean shit to the average end user. only programmers care if they have access to the source code, and most people aren't programmers. and i am a programmer, and i don't give a shit. the freedom to not think about my operating system and just get work done overrules all of those for me, so i use windows.

There are a few misplaced assumptions made. The easiest to refute is that all programmers are the same — if the author doesn't need the sources for something, neither should you. While there are few people who, say, need source code for their operating system, the small number of those people should not suggest that the sources are in fact useless. A builder usually would not find much use out of a sawmill, even if the job of the builder is to "cut things into shape". While such tools are probably irrelevant to a builder and application programmer, they would both be unable to do anything if someone didn't have the tools.

The most concerning assumption made is that most people will never program, so access to source materials is unimportant. It is difficult these days to genuinely argue that people should consider learning to program; in part because there is higher demand for programmers, as well as engineers, scientists, and generally all that sort of industry, as more industries make use of automation and more software, and so all calls to "learn to code" seem to come from industry demands.

But there are reasons to program which we would consider purely selfish. One reason, as outlined in Terminal Boredom and possibly originated in Goldberg and Kay's Personal Dynamic Media, is that the computer allows for the creation of methods of self-expression (i.e. meta-media). If we are to believe the idea of "linguistic relativity", then meta-media makes it possible to indulge in new modes of thought, which would be very powerful.

If not, we can at least make modes of thought that we already know a lot easier to engage in. Where a computer connected to some machinery can (partly or wholly) automate boring manual labour, a computer by itself similarly automates thinking labour. Minsky discusses the idea in Why programming is a good medium for expressing poorly understood and sloppily-formulated ideas, mostly relating to symbolic AI problems, and Iverson and Lamport attack mathematical proofs in Notation as a Tool of Thought and How to Write a 21st Century Proof respectively.

The "merely" mathematical domain of these articles does not induce much confidence that computing is useful for the creative spirit, but it can be explained by the observation that, generally, computers have been either inaccessible due to their scarcity, and have only been used for profitable purposes, or most people who have access to computers have not considered programming them, and so haven't written about more interesting uses. The assumption that programming is boring can only help itself: someone who believes this as an essential trait can only ever create a culture of boring programming.

Instead, what the assumption results in is a reaction towards a "human scale" of computing. But this ends up in a waste of resources and time; a computer is used to assist with reasoning about things, which provides the user with a means for more agency and subjectivity, but the computer is only used for mundane tasks which the authors of the software believe is only what the user can handle. In other words, despite the assistance provided by the computer, this line of thinking ends up assuming the user is always an idiot.

The programmer better be made an idiot

There is also the assumption that programming is hard, so there must be specialisation and a division of labour surrounding it. It is amusing to consider that many proponents of this sort of "post-open source" promote dubious and bad programming techniques.

The first instance we encountered was the introduction to the so-called "Anti-Capitalist" Software License, which contains the paragraph (emphasis ours):

> ACSL recognizes that the copyleft requirement of open source can be a drain on limited resources, can **expose sensitive or secure information**, and **can put software at risk of theft**. The ACSL allows integration into closed source projects such as games, security tools, artworks, and personal projects without requiring those tools be made public. **The availability of source code is less important than the organization of software labor.**

The second emphasised phrase makes no sense: how can software be stolen if you have allowed anyone to use it for whatever purpose? We will get to the third eventually. But more relevant now is the first phrase; there is the assumption that publishing source code will somehow undermine the security of a system.

To put it politely, your system is completely broken if publishing source code is a problem. Auguste Kerchoffs wrote six design principles for secure ciphers, one of which being "The system should not require secrecy, and it should not be a problem if it falls into enemy hands." Claude Shannon used the same principle, designing under the assumption that "the enemy knows the system". While there is no encouragement to write insecure systems, the existence of the license is apparently predicated on bad security design.

In the aftermath of A Parastatal Problem, one of the authors of Parastat wrote that their usage of C is equivalent to using a different model of car:

> I wish tech people would handle technologies and programming languages like Regular Car Reviews handles cars — different cars suit different people, and every car has the potential to be useful to/loved by someone.

We hope that the reader is wise enough to not consider hopping into a model which is known to spontaneously explode.

The author of the "Cooperative Source License" (note the middle word is Source, not Software) finds writing good code to be a fool's errand:

> One last thing about DRY and legacy code…
>
> They're just going to find a reason to hate your code and rewrite it anyway so stop fussing about it.
>
> They don't give a shit about your abstractions, your DRY, or your brilliant design.
>
> "Beautiful Code" and any such similar sentiment is a masturbatory fantasy for neckbeards. Don't buy into it.

It is hard to tell whether this quote provides real advice given in good conscience, or if it is someone's internal monologue while they push themselves to barely finish something before a deadline. The introduction to A Parastatal Problem argued that what was presented was "the norms of the corporate world" and "the development hell that one would expect from a lousy startup" — such descriptions are no less applicable here.

It is also consistently unfotunate to see fellow radicals and queers call everything that they do not like a product of "neckbeards" or "techbros", regardless of whether the reality is strikingly different. We normally see this as a means to avoid investing energy in people who are generally a waste of time or who are arguing in bad faith. Now we see it being applied to evade different ways of thinking and valid criticism, regardless of whether the thought is in reality of a queer nature or origin.

What's the deal?

But why make programming hard? Robert Strandh argues in The psychology of learning that, in the context of learning, some people are perfection-oriented, in that they search for new techniques and find the process enjoyable; and others are performance-oriented, in that they rather take productivism to heart and want to achieve immediate performance at all costs. It is reasonable that the perfection-oriented people would out-perform the performance-oriented people by performing self-improvement, so the latter find the need to "discredit [the former's] knowledge" in order to reduce the performance difference.

It is possible that a similar technique is used to keep post-open source relevant in some ways: if programming was easier, and if it was even enjoyable, there would be less of a need for a division of labour, and so there would be no need for the more rigid forms of organisation that its proponents promote (which we will discuss later). By repeating the lie that programming is essentially hard, people can be convinced that they need rigid forms of organisation to get things done. The issue lies in the subjective and contextual nature of what is considered "complex" or not.

One model we have heard of is that a task has some small "necessary complexity" and a typically larger complexity induced by poor tooling. But what complexity is "necessary" is only vaguely defined, and is more or less a projection of how hard the programmer thinks the task is. If the tooling improves, this projection changes, and so there wasn't really a notion of necessary complexity to start with.

Instead, we prefer to consider a good tool as something which makes complexity comprehensible and reasonable. For example, we may see a system we need to study as something with complex behaviour, or as the interaction of actors with much simpler individual behaviours. The difference in "complexity" again is exponential. A good tool would allow the computer to handle the toil of reasoning about the resulting blow-up of scale in the *distributed system* we have invented, while the user specifies behaviour and requirements in terms of the simpler actors involved.

A computer-free example of this subjectivity is presented in a presentation by Alan Kay, wherein some primary school students are given a fairly laborious task of cutting out paper shapes in order to make larger and larger ones, and count how many smaller shapes were used. The teacher explained that the purpose was to "slow [the students] down so they'll think" — the students eventually worked out the rate of change, or rather the derivative, of the number of smaller shapes, and integrated it to find the actual number of shapes used. When the students had this perspective, the result they produced was "a second-order discrete differential equation, derived by six-year olds." Someone who did not watch the demonstration, and only heard the conclusion, would think Kay was telling a joke.

When equipped with poor tools for the problem, a problem appears much more complex than it needs to be. Such perceived complexity presents many challenges: a post-open source commons is unequipped to compete with any corporate and proprietary vendors; and it forces people to associate in order to get anything done. This coercion does not lead to pleasurable or productive associations.

Legalese or a liberatory program?

The proposed relations by post-open source advocates fetishise the community in some way or another. We have already noted a similar issue in the Fediverse within *Careful with that axe, Eugen*; where a server is supposed to be a community. This squares social circles, so to speak, and turns free associations into discrete "communities". (While many people seem to have come to the same abstract conclusions, and advocate the same organisational strategies, we will pick and discuss specifics from the so-called cooperative source development process.)

Whereas the barrier to association in the Fediverse is technical – one can only be associated with one server at a time – the barrier in post-open source is monetary, as it is expected that people will pay to get into the "community". This pay wall is supposed to make development "sustainable" by providing funding for the producers.

Once in the community, there are supposed to be democratic votes on things, which is supposed to allow consumers to have a say in development. However, with the combination of pay-walling and that the original author was around first, it would not be difficult for the original author to kick out any dissenters quickly. As we had noted with proprietary software in *Ethical software is a sad joke*, someone who disagrees with the way development is done would find themselves completely unable to use the software, let alone able to modify it themselves; is this really an improvement over being told to “just fork it” and maintain a copy yourself?

A programmer today has an interesting role of being a producer *and* consumer of the same products; specifically, they tend to re-use other people’s code while producing their own code. When one goes to find code, they typically look up the concept they are looking for, pick the best few options out of how many there are, and then skim the code and documentation to see if it would be useful to them.

Unbeknownst to the programmer, they are participating in a form of association with very low overhead in many ways. The matter of finding code can be done in ten minutes or so, and the authors of that code do not have to advertise or run a business or anything to find use; the openly available documentation and code speak for themselves.

When collaboration is performed over the Internet, it is possible for the individuals involved to be in very different time zones, and thus any communication could require waiting some time, until enough of the participants are awake. No communication is necessary with free code reuse — one can take the code and continue working immediately. However, if the code is only available after asking to enter the “community”, there could be a delay of many hours, if not days. And, as attestable by many programmers, the creative spirit is an impatient one; such association would require far too much patience if one is burning to hack on an idea. A prerequisite for development of anything to be “sustainable” is to have development at all, and it is hard to say that participants in such associations would be very productive.

We also hope that “post-open source” people understand the name means “open source” has already happened — is there even a chance to compete with open source, if the barrier to reuse is significantly higher with phony “communities”, which are all more or less a fancy name for one’s income stream anyway?

What did we learn from the old free software?

The kinds of associations formed by many users of the old free software have many good properties. For example, association and dissociation of individuals can be fast and can be done cleanly; when we look at relations between individuals rather than large, static organisations, there are many more ways to make relations between individuals, and so there is less of a reason to associate unless one benefits from it.

Let’s look back at Stirner’s Critics for a bit:

> If Hess attentively observed real life, to which he holds so much, he will see hundreds of such egoistic unions, some passing quickly, others lasting. Perhaps at this very moment, some children have come together just outside his window in a friendly game. If he looks at them, he will see a playful egoistic union. Perhaps Hess has a friend or a beloved; then he knows how one heart finds another, as their two hearts unite egoistically to delight (enjoy) each other, and how no one “comes up short” in this. Perhaps he meets a few good friends on the street and they ask him to accompany them to a tavern for wine; does he go along as a favor to them, or does he “unite” with them because it promises pleasure? Should they thank him heartily

> for the "sacrifice," or do they know that all together they form an "egoistic union" for a little while?

With these properties in mind, it does not seem far out to call the associations made "unions of egoists" or "affinity groups".

Earlier we had promised to investigate the statement "The availability of source code is less important than the organization of software labor", and we are in a position to do that now: what we find out is that the published source materials are effectively the means of "organisation" in themselves. Any more organisation is plainly unnecessary, and trying to separate the two purposes of source materials, as they are used today, results in utterly absurd reasoning.

Is such (lack of) organisation "efficient"? Is it even "sustainable"? One may as well ask where the "communist countries" are, or some boring crap like that. But we can give some general ideas. Good programs generally are the product of *dogfooding* — the producer also consumes their own software, and can directly judge it and figure out what needs to improve. For example, someone who draws would have an immediate idea of whether the drawing program they made is any good or not, but a programmer merely contracted to make a drawing program might only check off a list of requirements, and the programmer is at the whim of the people involved in making requirements to produce relevant requirements. So someone who "gets" their domain is going to have a better chance of success than someone who doesn't get it; a self-interested programmer is the most efficient at writing software they want to write.

The empowerment of "consumers" also provides for better quality software. Using our previous principle that a creative spirit tends to be impatient, it may be better for one to fix deficiencies in software, rather than file a request to have it fixed by someone else. Again, if people wrote better code, the effort to modify it would also be reduced. The Fuzz Revisited paper seems to support this hypothesis; the commercial Unixen tended to be more bug-prone than the equivalent GNU and Linux software, and the latter two could be fixed by anyone who found a bug.

But a usual retort would be that the people who fix software when they find bugs are still programmers, and not ordinary people — this couldn't work with anyone who isn't a programmer. So do people tend to write good code? There may seemingly be some effort involved in writing good code, but it is far from wasted. As Robert Strandh notes on "continuous improvement", careful application of efficiency-improving techniques means "[he is] able to use more time eating, sleeping, and relaxing." And the "great virtues of a programmer", at least according to Larry Wall and friends, of laziness, impatience and hubris are merely part of an egoistic affair. So it is not unlikely that, if someone had any egoistic motivation to program, they could be a great programmer.

Another interesting property of the old free software was a sort of edginess performed by the individuals associated with it. The stereotype of a hacker cussing out Microsoft under every breath is unfortunately the best manifestation of such edge; the hacker perhaps has a vague idea of what corporates have done to mess with the possibility of a software commons, but hasn't put a name to it, and is basically conditioned to never use their observations to produce any anarchic concepts.

We can't help but ponder what similarities exist between, say, that hacker remaining cautious of the new Microsoft and its "Microsoft <3 Linux" campaign, and the queer anarchist reminding their friends that the new gay politician is not the end all in queer liberation. Of course, there could be nothing similar at all, but both groups have succeeded more than others in preventing the corporate recuperation of their domains.

In order to avoid such edge and nearly political endeavours, someone had the wise idea of inventing an “open source” movement which reduced the emphasis on a software commons. While the post-open source “movement” has added in its own political endeavours, which appear radical at face value, the anti-commons sentiment remains.

What is egoist telekommunism?

We consider egoist telekommunism to be two things:

It is an abstract free software, but groups like the Free Software Foundation despise it, as we find commercial involvement in the commons to be highly dubious. (Of course, the FSF is bad at making decisions generally, such as when they opposed the disallowment of using software in organisations which can’t even follow local labour laws.)

It is also a synthesis of egotistical reasons to want things, and telekommunist ways to get things. An individual who wants to do “producer” things would appreciate being able to reuse public resources, and that like-minded individuals provide their own improvements; and an individual who wants to do “consumer” things would find it more empowering to tweak their software themself, rather than have to find the perfect developer who has exactly their interests in mind.

Our notion of egoism coincides nicely with telekommunist ideas. For example, the telekommunist rightfully critiques a concrete form, critiquing cooperatives as exploitive and bureaucratic, and similarly the egoist critiques its general shape, as an insular community wherein its participants cannot fulfil themselves.

With such a focus on general principles, it is easy to see that our prior writings attacked the same general ideas, specifically, the necessity of a distinction between producer and consumer (A Parastatal Problem, “Ethical software” and this article), that computing must necessarily be hard (Terminal Boredom and this article), and that rigid “communities” are a good representation of associations (CWTAE and this article).

Some people have figured out their own critiques of some of these ideas, but still fall short and fool themselves with the others; and some people only handle instances of these ideas and not the ideas themselves, leading to an inconsistent view; for example, one might correctly dismiss a concept of proprietary “cooperative” software, but still support the production of more isolated communities on so-called “social” networks.

One is foolish to merely blame their problems on capitalism, especially when their idea of “anarchism”, or “anti-capitalism”, or whatever else they hold dearly, reproduces the same logic and oppressive practices. There is no concrete “existential threat” to peer production, there are merely false principles, and any exhibition of such principles is no less terrifying than any other. The notion of “post-open source” as we now know of it, much like how we have said before, was dead on arrival.

Gnuxie Lulamoon and Hayley Patton
The Poverty of Post-Open Source
September 4, 2021

Retrieved on September 10, 2021 from
https://applied-langua.ge/posts/the-poverty-of-post-open-source.html

theanarchistlibrary.org