

Open Source Software: Altruism?

E. M. Recio

PHIL 251: Ethics

Department of Philosophy
College of Arts and Sciences - Drexel University

June 13, 2000

Abstract

Altruism in the computer software industry is not only advantageous but clearly necessary for the production of useful and bug-free software. Altruism is the core component of the Free Software Movement. It is based on the simple principle that software is an intellectual creation and cannot be kept behind closed doors if it is to prosper and become useful. Through the GNU Public License (<http://www.gnu.org>) the Free Software Foundation has been able to distribute and keep free the flotilla of software currently in use-transparently- throughout the world (more so than Microsoft's software.) As such, secrecy and obscurity has no place in the software industry, and is detrimental to the advancement of computing technology.

For years the sciences have sprung up unhindered by people just asking questions. Maths and Philosophy, being at the root of all modern sciences, have been able to lead the way into unexplored territory. It is time now to return to philosophy, in order that we may be able to resolve a new set of problems, which seem to be brewing in the face of globalisation, revolutionised by a medium, unlike any other since the invention of the printing press, the computer and the Internet.

1 Behind closed doors

1.1 Status Quo

Let's take a look at the current state of the software industry. It seems like big companies, such as Microsoft are thriving under this technological boom. Computers, indeed, have come a long way since their old counterparts in the late 1970's and the software right along with them.

Now-a-days people flick a switch (some not even that, just pressing the space bar) and their personal computers come to life. Flashy logos and trademarked names adorn the monitor as the user waits for his or her operating system to bootstrap itself. After an interminable wait, and the abatement of crunching sounds, mystically emanating from a small box next to them, the user is ready to carry on their intended function with this modern day swiss army knife.

The user then proceeds to read his email, browse the world wide web, and download his favourite song. He orders a pizza, makes flight reservations,

pays his bills, and even makes a quick free telephone call using this tool. He never leaves his desk; never talks to anyone; and it is all seamlessly integrated. Everything is going fine, until, for some unknown reason, he gets a message from this tool stating that it has done an illegal instruction.

The user simply confirms the dialog. And if he is lucky, can ask the tool to re-boot itself to correct this *so called error*. If he is not so fortunate, the user must press the reset button, and in some cases remove the battery from his portable tool, for this computer has *locked up*.

1.2 The Shift

The scenario above denotes a major step in the development of technology. Technology has been moved from the periphery to the centre of our web of thought. I even find myself asking someone I just met for their email address rather than their telephone number. If you would turn the way back clock to five years ago, this would seem a rather awkward happenstance.

1.2.1 Coping

Since technology of this sort has moved to an everyday occurrence, we are rarely aware of what happens behind the scenes. In fact, we *cannot* be aware of every small detail if we are to use the technology effectively and efficiently. However, in the closed source world, we are *unable* to just look under the hood to see exactly what is happening.

This powerlessness is the very reason why we are prey to such failures in computer technology, and yet continue. To compensate for our incapacity to fix the program, we cope and rationalise. We may rationalise the failure in several ways:

1. “computer programs are bound to have bugs and we should just deal with it.”
2. “the computer has helped me in x ways. This is just a price we pay.”
3. “they are still working out the bugs, I’ll buy the upgrade to fix it.”
4. “I really don’t have much of an option, I’ll just wait for the next version.”

There may be many variations of these four statements, but their rationale all conclude one or both of the following: buy a newer version (hoping that the newer version fixes the old problem); deal with the issue by ignoring it.

In order for the computer user to embrace Open Source (along with the implications of altruism,) they must go beyond these types of excuses for technology failure. These four arguments above can be addressed in the following ways.

1.2.2 “Computer programs are bound to have bugs and we should just deal with it.” and “The computer has helped me in x ways. This is just a price we pay.”

This is utterly unacceptable in today’s world. Spending money for a piece of software implies that you are getting something of utmost quality. The \$300 USD that is forked over for Microsoft Office 2000 Professional, for example, ought to mean that it’s worth \$300 USD worth of someone’s labour. In so being, it implies that this person (or company) has given their seal of approval which states that the software is of high quality and extreme effectiveness.

Unfortunately, the \$300 USD that is spent on this piece of software is exchange-value only. You are not getting effectiveness, and efficiency. You are getting what the company thinks you will most likely pay for, nothing more than bells and whistles. You are paying for the number of features in the software package, features which, for the average user, remains untouched; you are paying for the affable “paper clip” (called Clippit) to come down and tell you what you can and cannot do. This, again, for the average user, supplies little or no useful information. If you pay \$300 USD for a piece of software, at the very least, it ought do what it purports without a hitch. This, as any computer user will surely tell you, is rarely the case.

Drawing a simple analogy between the automobile industry and the computing industry may help to demonstrate my point. Consider buying a new car; after only a month, the car begins to stall every so often. You must restart the car in order to continue driving. Using the above arguments applied to a car, we have: “cars are bound to have bugs and we should just deal with it,” and “The car has helped me in so many ways. This is just a price we pay.” The absurdity in the argument is quite clear now. It’s unacceptable for automobiles, why then should we accept it for computers?

1.2.3 “They are still working out the bugs. I’ll just buy the upgrade to fix it,” and “I really don’t have much of an option, I’ll just wait for the next version.”

Again, this is quite unacceptable now-a-days. You paid a particular amount of money for a some piece of software *because* you expected the software to behave in a particular manner. You needed it for typing your thesis paper, or organising your financial records. There is no reason why the software should not perform as advertised or be substandard.

Software package upgrades oft times have additional features built upon the old features. Rarely is a software package upgrade written from scratch. As such, software that is not re-written from scratch or, software that is patched to fix bugs and holes creates what is known as *bloat ware*. Bloat ware is a term endemic to the software industry which means that the more features added, or code which is fixed, the bigger the program gets, with a notable decrease in performance. From the programmer’s point of view, such bloat ware can become a nightmare.

Consider Netscape, the once popular world wide web browser. In an attempt to release their new version of the browser (version six) Netscape programmers came across the dismal realisation that their software had to be rewritten from scratch due to the amount of bloat ware already in existence in the product. Their original timeline decreed a one year development cycle before their first release of the new version. It is now two years and counting. They have (as of yet) to release a stable version of the new browser.

The car analogy above may be extended to include these two arguments for the *coping* of software failure. Consider our car that stalled randomly every so often. If we were to use the same rationale for the automotive industry as we use for the software industry, then we would be voluntarily taking our cars to mechanics to fix the automobile (spending our own money in doing so,) so that we could get our problem fixed. At the same time, the mechanic would install a four wheel drive apparatus in the trunk, replace our tape player with a CD player that sticks out prominently. So that now, the car doesn’t stall as often anymore, but is sluggish and doesn’t perform as well as before, due to the increased weight and design of the new addition of hardware.

2 Strictly Speaking

Before we continue, we must first attempt to understand some of the underlying concepts involved in the open source movement. Most importantly we can concentrate on the “oughts” of the software industry as compared to everyday life.

Furthermore, it will be evident that open source doesn’t necessarily imply wanton altruism. In the open source movement you can have vested interests in any particular project. Meanwhile, these interests can *still* benefit the software industry by making it open source.

2.1 Ethical Egoism

Ethical egoism attempts to answer the question of what *ought* to be done. As such, it states that everyone ought to seek out their own self-interests; if you should happen to help another by seeking out your best self-interest, then this would be by accident.

There are three major arguments for ethical egoism:

- (1) By looking out for others’ interests we are insulting them; may do more bad than good and; may intrude on others’ privacy.
- (2) By looking out for others’ interests we are denying our own individuality.
- (3) All moral duties have an underlying principle of self-interest.

2.1.1 Adding Injury to Insult

The first argument for ethical egoism states that looking out for others’ interests is self defeating. The argument may be stated in different ways but it boils down to the following three ways of thinking.

We all know what our own interests are, better than anyone else does. By looking out for others’ interests we may be doing more bad than good, because we don’t know exactly what they want. Another way of thinking is that by looking out for others’ interests we are intruding on their privacy. And finally, by offering help (looking out for others’ interests) you are essentially stating that they cannot do the job themselves. Hence, it may be interpreted as an insult, because it robs them of the ability to help themselves.

2.1.2 One Life to Live

The second argument is that altruism leads to a denial of the value of individuality. It states that the ethics of altruism is not about to live life, but how to sacrifice it.

It all boils down to the following way of thinking: a person has one life to live. If we value the individual, then this life is most important. Altruism sees the life of the individual as something to be sacrificed for the good of others. As such, the ethics of altruism does not take into consideration the value of the individual human life. Whereas, ethical egoism *does* take the value of the individual life into serious consideration.

2.1.3 Fundamental Principle

Lastly, this line of reasoning consists, not of debunking current *commonsense* morality, but of providing a unifying, and underlying theory *for* them. This argument sees ethical egoism as a *fundamental principle* of all moral duties.

For example, we have a duty not to lie. Why? Because if we lie, then we would be seen as untrustworthy, and people would avoid us. Oft times we will need people to be honest with us; so if we lie, they feel as if they have no moral obligation to be truthful. It's basically in our best interests to not lie.

2.2 Software Do's

What does this all mean from the software industry's point of view? Many people working for big corporations, huge software industry houses believe that this ethical egoism is the "oughtness" of the computer software industry. Closed source proponents believe that it is in our best *self-interest* to produce closed source; that they won't make any money unless the software source code is not readable by anyone other than the people who produce the software.

2.2.1 Adding Injury to Insult

Let's address the first argument for ethical egoism as it pertains to the software industry. If you can recall from above, the first argument implied that we would insult, invade the privacy of, and injure individuals by looking out for their interests, rather than our own.

Are we really doing that when we ask people for comments? Are we really doing that when we *freely* distribute the source code? When we encourage individuals to look through the source code and contribute patches that change the software to better suit their needs?

A more inherent problem with this line of reasoning, though, is in the basic argument. Even though we are *acting* like egoists, we are actually displaying beneficent motives. The argument states that it's in everyone's best interest if we just but out.

2.2.2 One Life To Live

The second argument holds no water for closed source software designers in that we really aren't denying the individual at all. In fact, in open source development, we increase the value of the individual by allowing each person to view the source code, and we request their input.

If the individual does not like an aspect of the program, he or she may contribute changes to the software product to suit their needs. If you don't like the OK button where it is, you can change it, or have it say Alright instead.

Furthermore, closed source software houses ignore the individual. Do you know the name of the head programmer of the Microsoft Windows kernel? In contrast, many of the developers of the software products under linux are very well known (to the community.) Oft times, the email addresses of the individuals responsible for various aspects of a software product are in plain text, noted in various places, in the actual running program. Closed source anonymises the individuals.

2.2.3 Fundamental Principle

Finally, closed source doesn't support basic moral duties. By open sourcing our software, false claims, harm, and other such moral don't are not supported. In fact, open source software allows for us to follow a duty like "ought not lie" because then we don't present false claims as to what a software product does in order for someone to buy it.

Open source software does no harm by providing an open method of all eyes to seek, identify, and fix security holes, and bugs. In fact, because of all of these things, it is in everyone's best interest that software be open sourced. Basically, everyone benefits from it.

3 The Difference

3.1 Blaming the Victim

The software companies, by forcing the user to upgrade, and forcing the user to take a passive stance on these issues, implies that the end user is responsible. Even with bug fixes that are free, require that the user actively find and download the patch. They take no responsibility for the bugs, and most companies even state, in their license on the technical support page, that they are not legally liable for these bug fixes (as they pertain to the ability to fix a problem.) Sometimes the closed source operating system will display a message stating that you should get in touch with the vendor of the software that crashed (when it was the operating system itself that crashed.)

With all of this finger pointing, and blame throwing, no one stands up and takes responsibility for their actions. Consider the car analogy again: if the car were released from the factory with various defects (ie: loose bolts on some gear in the engine.) The company would publically state that there was a defect, and that they would exchange or call back all models of a particular type that was sold. They acknowledge the fact that *they*, indeed, were responsible for the “bugs” in the automobile.

3.1.1 Closed Source Case Study

The computing industry has, as of yet, to take this step. Consider the security hole found in Microsoft’s FrontPage webserver software. This software product was in direct competition with Netscape Communication’s webserver.

They released this software in 1997. Many companies used this software to serve their web pages for years. Many ISP’s and major corporations are still using this software. The software designers intentionally placed a backdoor into the webserver. The password for the back door was, no less, “Netscape engineers are weenies.” More importantly, the backdoor was discovered in April of 2000, over three years after the release of the software product.

There are three elements for alarm in this incident. Firstly, the software engineers of this product, for whatever the reason, thought nothing of the major *implications* of a backdoor as a security hole to a publicly accessible webserver. Secondly, the nature of the production of software, closed source, meant that this backdoor would be able to exist for long periods of time. Lastly, the speed at which the company needed to release the software

product to remain competitive, cost the scrutiny of the software product.

Microsoft, to this day, will not admit that there is, indeed, a backdoor. But ironically enough, they have published a webpage stating how to remove this non-existent backdoor. I will examine these elements after the following juxtaposed case study.

3.1.2 Open Source Case Study

Shortly after the discovery of the Microsoft backdoor to their webserver, an Internet security company discovered a backdoor in an administration tool specific to a popular open source Linux distribution, Red Hat. The administration tool, codenamed Pirhana, used to configure, and grant unhindered access to the Red Hat Linux operating system, had a backdoor.

The password for this software was a simple “Q”. Before the news even hit the websites, the company Red Hat, had released a source code patch for the software to fix the problem. They had confirmed and acknowledged it’s existence. The response was timely, and efficiently executed. They even offered a reason for the existence of the backdoor: to test the software before it was released. Red Hat also stated that this backdoor was immediately closed when the system administrator would remove the sample user configuration files from the system. However, this note, was buried in the documentation, and rarely the system administrators read to that depth.

This Pirhana tool, had been out no more than two weeks to a month before the security hole was discovered. Furthermore, the Internet security company discovered this backdoor by closely scrutinising the *source code*. Surely, had the software been closed source, it would have taken *much* longer for the backdoor to be discovered.

3.2 Taking the Blame

These two incidents may seem similar to the average user. There are obvious differences between the open source response and the closed source response to these “similar” incidents. The open source community embraces criticism. The closed source community does not. Ultimately, the company is to blame; Let us dissect the “critisizing” implications on the software industry.

3.2.1 Software Methodology

The basic premise behind open source: given enough eyes, all bugs are shallow. The free software community openly distributes their source code with the goal that if enough people are critical of it, some of them will fix the problems with it. Or at the very least, critique the software package, point out its flaws, &c. This is called the *economic principle of abundance*. Ironically enough, even though it's free, it does not imply that you get what you pay for.

The basic premise behind closed source: profit. By controlling the means of distribution of the software (adopting an *economic principle of scarcity*), the company that is producing the closed source product can profit from it. They promote security through obscurity. But eventually, as has been demonstrated in the past, someone, somewhere will figure it out: Consider the DVD's encryption scheme, which was broken in less than a year, by an open source movement programmer. As such, the companies would attempt to keep quiet many security risks to their system (as it devalues the product's and company's credibility.)

3.2.2 Instinctual Reactions: Speed, Efficiency, and Reliability?

Considering the software methodology of both the open source movement, and the closed source movement, it comes to no surprise that the instinctual reactions of each were made such that they supported each's view on software.

The closed source view was to promote their economic principle of scarcity. They did not openly admit to this major oversight; the backdoor was discovered *years* later; and they did not react in a speedily manner. Doing so, would admit to users that the bug was indeed an issue to be addressed, and trust in that company's software would diminish.

4 It's just a job... but is it?

Here is where altruism would have a large role to play in the software industry. There are many such examples as stated above. Two camps may be delineated in the production of software: those who do it for a job, and those who do it for fun. As in other fields of work, those people who do it for fun, oft times do a better job at it.

4.1 The engineers from the enthusiasts

4.1.1 Self-Interest oriented developer

Those software designers and engineers are often under great pressure to produce software under strict deadlines. The reason for this: competition. This pressure to produce a lot in short amount of time, means that the quality of software suffers. As a result, the end user must pay for the mistakes made by programmers.

Software companies often do not know the true wants and wishes of the end users. They keep a project very quiet, and secretive, while they develop it. And when they are done with the software project, they unveil it. It has too many features the user doesn't need, and none of the features that the users need.

These people also live 9-5 workdays. Work is work. This is what they get paid to do, whether or not it works right is not their problem right now. They will fix it after the bug is reported by a user. The point of coding is to serve their own self-interest: profit.

4.1.2 Altruistic oriented developer

The *altruistic* open source software developer has a different approach. He releases his software's source code, and final product often and early. In this manner, bugs are detected *before* the software product is released for general use.

The open source developer is constantly requesting feedback from users *and* programmers. By releasing the source code before the product is complete, allows for a positive feedback cycle. If the critic sees that his changes were incorporated into the product, then he would continue to use and even contribute features to the product. Otherwise the critic can branch and start his own software product. This branched product may be similar to the original, but incorporates the changes that he wants.

Consider the case between Emacs, and XEmacs. Emacs is a UNIX text editor. It's a feature rich software product. One day, there was a disagreement between the developers of this open source project, as to what features ought be supported. Now we have two competent software packages which concentrate on two very different environments: the console (or text based environment) and the graphical user interface (or a windowing environment.)

5 Putting it all together

Most of the open source projects are based on one thing: creating a better software package. Unlike the software engineers bent on profit, the open source developers believe in standards and methods that will benefit the programmers, and end users alike. They start, develop and end a software product with the users in mind.

Closed source companies and closed source developers start, develop and end a software product with profit in mind. They care about user's input, only so far as the minimum work done to keep a user as a customer is necessary. This often means buggy software, relying on the users' naivete, to upgrade their software *at their cost*.

The altruistic programmer considers public concensus. Ultimately he has final say on the product, but the public can branch their own version if they are unsatisfied. This fosters competition, which leads to creating a better product. If all software companies would put the user first and not worry so much about profits, we would have very stable and efficient commercial software packages.

Altruism: putting others before you, would produce much better software.